Empirical Studies of Novices Learning Programming

Thesis

Empirical studies of novices learning programming

Ann Carolyn Jones (B. A.)

Submitted for the degree of Doctor of Philosophy in Educational Technology at the Open University, Milton Keynes

October 1989

EX12

HIGHER DEGREES OFFICE

LIBRARY AUTHORISATION FORM

STUDENT: Ann Jones SERIAL NO: M7019923

DEGREE: PhD

TITLE OF THESIS: Empirical Studies of Novices Programming (check exact title!)

I confirm that I am willing that my thesis be made available to readers and maybe photocopied, subject to the discretion of the Librarian.

SIGNED: [signature] DATE: 11th September 1990

Abstract

This thesis is concerned with the problems that novices have in learning to program: in particular it is concerned with the difficulties experienced by novices learning at a distance, using instructional materials which have been designed especially for novices. One of the major problems for novices is how to link the new information which they encounter with their existing knowledge. Du Boulay, O'Shea and Monk (1981) suggest helping novices to bridge the gap between their existing knowledge and new information by teaching via a conceptual model, which serves to explain the new information in familiar terms.

In this thesis the difficulties which novices have when learning to program with the help of a conceptual model were investigated. The curricula and conceptual models of four different programming languages are examined, all of which were designed to teach novices. Du Boulay, O'Shea and Monk (1981) have suggested criteria for analysing conceptual models. It is argued that these criteria, however, do not address the presentation of the conceptual model, and so are insufficient to evaluate them. An additional form of analysis was proposed and used, in addition to the criteria offered by Du Boulay et al. This is a way of describing the conceptual model which distinguishes three views of the conceptual model: state, procedure and function, and which highlights the different aspects which are important for the novice learner by identifying the different kinds of knowledge which are necessary to understand the conceptual model. This analysis of the conceptual models showed that the environments are not as exemplary as the du Boulay et al's criteria suggest, and indicated that three of the environments, SOLO, PT501 and DESMOND, lack a functional representation, and that the fourth, Open Logo, has other different problems.

An empirical study was carried out to study the transfer effects of learning two of the languages, a high level and a low level language, sequentially. There was no evidence for such transfer effects. The difficulties novices have in learning the four different languages were also investigated. These studies show that even though the novices were studying environments designed for novices learning at a distance, they did not develop good levels of competence, and the problems they had fall into two main categories: programming and pedagogical.

Although the different languages had different aims and curricula, novices had some problems which were common to all or most of the languages. These included understanding flow of control, developing and using programming plans, developing accurate mental models, and in the high level languages, understanding recursion. It is argued that some of these problems are related to the conceptual models. In particular, the difficulties novices had in developing and using plan knowledge, which is one of their main problems, can be explained by the lack of an appropriate functional description in the languages.

The subjects' pedagogical problems arose from the relationship between the style and structure of the curriculum, its content, and the subjects themselves. In all the four texts the teaching material is very carefully structured and it is suggested that this may encourage the learner to adopt an over-dependent attitude towards the text, and in some cases, to work at an inappropriate syntactic level.

The relationship between the distance learning situation and the novice programmer is discussed, and recommendations are made for improving the curricula used for teaching novices programming.

Dedication

To the memory of my father

Acknowledgements

My greatest debt is to my internal supervisor, Tim O'Shea, without whose encouragement and support the venture would not have got started, and without whose advice, patience and good humoured nagging, would certainly not have been finished. My thanks also to my co-supervisor, Thomas Green, who provided constructive criticism and encouragement at just the points where they were needed.

The Institute of Educational Technology has provided a friendly and helpful environment in which to work. I would like to thank Olwyn Wilson, Hansa Solanki and Dave Perry for their practical and technical help, and all those members of the Institute who helped the difficult task of doing such work part time, and understood the conflicts of interests: in particular Eileen Scanlon, David Hawkridge and Diana Lauriallard, and Tim again. Several members of the Centre for Information Technology in Education helped with particular stages of the work: I would like to thank Claire O'Malley and Alistair Edwards who evaluated the Logo tutorial manual, and Pat Fung who also did this and gave me valuable comments and practical help. I am also grateful to Diana Laurillard and Mike Sharples who read and commented on the thesis and to Laurence Alpay who proof-read the thesis. I have also had very helpful discussions with Mike at earlier stages of the work.

The CAL research group has also provided a supportive research base. In particular I have benefitted enormously from discussions with Peter Scrimshaw and his comments on various chapters. Thanks also to Jenny Preece for her encouragement and comments. I would like to thank all the people who have not been mentioned by name but who have provided support and help of various kinds during this period.

I ran a joint experiment with Tony Hasemer, and would like to thank him for his support and encouragment in doing this, and also Marc Eisenstadt, who was particularly helpful in the early days.

I would like to thank Professor Mayer, Santa Barbara, and the staff at Xerox Parc Laboratories, Palo Alto, who gave me help and encouragement during a trip to the States, and also Richard Young and Ben du Boulay who have discussed my work with me at various times.

A particular thank you to Jim Brannen for his support and optimism and for commenting on a draft.

Finally I would like to thank all of the people who acted as subjects in the experiments and who took part in the study, without whom none of this would have been possible.

Contents

Figures

References

Appendices

Chapter 4

Appendix 4.1 Open Logo instructions
Appendix 4.2 The PT501 instruction set
Appendix 4.3 DESMOND's instruction set

Chapter 6

Appendix 6.1 The SOLO summer school project
Appendix 6.2 Some examples of the exercises in the SOLO manual

Chapter 7

Appendix 7.1 Initial Questionnaire
Appendix 7.1 Final Questionnaire

Chapter 8

Appendix 8.1 Concepts introduced in the Open Logo toutorial manual
Appendix 8.2 In-text questions from the Open Logo tutorial manual
Appendix 8.3 Evaluators' reports

Chapter 9

Appendix 9.1 In-text questions and answers
Appendix 9.2 Example comment sheet
Appendix 9.3 DESMOND plans
Appendix 9.4 Statistical analysis and quantitative data summary

Chapter 1

THE RESEARCH PROBLEM AND OVERVIEW OF THESIS

Contents

1.1 THE RESEARCH PROBLEM

This thesis is concerned with the difficulties that people have in learning to program when they have no previous experience, and with the problems that they encounter in using self-instructional or distance learning materials which have been specially designed for novices. The application of distance learning technology may seem strange in a domain with a large practical hands-on component, yet distance-learning methods are being increasingly used to cope with the demands of training in new technology. The Open University has responded to the demands of new technology by producing courses which use computers in a variety of ways. Such uses include: use as learning tools in a variety of subjects, - for example, tutorial computer assisted learning; as tools for production and design (e.g. word processing and spread sheets); and in teaching about computers in various ways, (for example in training courses aimed at teachers using microelectronics and in teaching programming concepts). The first motivation for this thesis, then, is the issue of the best way to produce effective material for students who are essentially on their own.

A major problem facing novices is how to link the new information which they encounter with their existing knowledge. Both Piagetian and artificial intelligence models of learning emphasise that knowledge must be structured in order to be meaningful and accessible. In order to learn, new information needs to be assimilated into the existing cognitive structure, (which in turn will undergo changes).

Du Boulay, O'Shea and Monk (1981) suggest helping novice programmers to bridge the gap between their existing knowledge and new information by teaching via a conceptual model, which serves to explain the new information in familiar

terms. An important role of the conceptual model is to help the user understand what is going on inside the machine, at an appropriate level. The user needs to understand the effects of commands. For example, in SOLO, the user needs to know that the NOTE <triple> command will add a triple to its data base and will also print it on the screen, whereas the PRINT command only prints the triple on the screen. The effects of the two commands are similar: in order to appreciate the difference the user needs to know how SOLO works. A conceptual model provides ways for the user to see the notional machine in action.[1] Du Boulay, O'Shea and Monk (op. cit) refer to this as a glass box approach, where the user tries to understand what is going on inside the computer. The idea is to describe the relationship between a command and the subsequent change in the state of the computer. Du Boulay et al offer two important prescriptions for making the hidden workings of a programming language more manifest: simplicity - there should be "a small number of parts which interact in ways that are easily understood" and visibility, that the novice should be able to view "selected parts and processes" of the model "in action."

In this thesis, the difficulties which novices have when learning to program with the help of a conceptual model are investigated, i.e. what happens under good, if not ideal, conditions? When designers have thought about and provided a conceptual model, what kind of mental model does the learner develop?

---

[1] The idea of a notional or abstract machine is referred to by several researchers and also used in this thesis. It is the behaviour of the computer when it is running a particular programming language described at the appropriate level: for example, in SOLO, when the user types in FORGET <triple> the SOLO notional machine deletes the triple from the data base. It is not necessary to know how the machine behaves at a more detailed level or what is stored in what register.

The curricula and conceptual models of four different programming environments[2] were examined, all of which were designed to teach novices. The problems that learners have when learning their first programming language, in forming mental models, making adjustments to these models, and in transferring these models to a new language were investigated by carrying out empirical studies. There is some evidence (Weyer and Cannara, 1975) that learning high and low level languages nearly simultaneously, facilitates the learning of basic programming concepts and one of the studies investigated the possible transfer effects between a high level and low level language.

In order to teach programming effectively, it is necessary to have some knowledge about how programming is learnt so that one can begin to understand the learning process at a detailed level. This knowledge can then feed back into the process of instructional design. The second motivation for this study, then, is an interest in learning processes. Programming is viewed as an example of a high level and complex skill, which is hard enough to learn to make it interesting.

1.2 OVERVIEW OF THE THESIS

Chapter 1

The remainder of this chapter summarises the structure of the thesis.

---

2 The term environment is used to include all the components that make up the environment in which the student is learning the language. This will include, therefore, not only the language itself and the teaching material, but also the operating system under which it runs, the documentation, error messages etc.

Chapter 2

The second chapter is a review of relevant literature. In addition to du Boulay et al (op. cit.) other researchers have also advocated presenting the workings of the notional machine in a way which will be familiar to novices, and which gives them a clear 'story' about the workings of the machine. Other studies reviewed in chapter two all bear on the problems that novices have but offer quite different approaches. The chapter is divided into the following sections: programming notations and language constructs; experts and novices; plans; conceptual and mental models; children, problem solving and field studies of novices' problems; learning to use complex devices; comparative studies of languages, and studies of LOGO and SOLO.

The final section summarises the findings reported in the literature and their relationship to this thesis. It points out that although much work has been done which is relevant to how novices learn using conceptual models, none of the studies address this issue specifically. The work reviewed on language notation is not directly relevant, and is often geared towards professional programmers, whilst this thesis is concerned with the problems that novices have. However, studies on programming notations and language constructs have led to design principles which are applicable to languages for novices. The literature on the relationship between natural language and programming languages indicates that there is no overall ideal first programming language which can be based on natural languages, and those which appear close to natural language, such as Prolog, attract their own special kinds of problems.

The work which is closest in its style and approach to this thesis, is the field studies of children learning LOGO and BASIC: in particular the work of Pea and Kurland

(1984), and also the work by Mack, Lewis and Carroll (1982) and Bott (1979) on text editing and word processing. What Pea and Kurland refer to as a developmental cognitive science perspective is similar to the perspective taken here, although the focus here is on adults rather than children. Following work on transfer skills, Pea and Kurland suggest two routes forward in this area. The first is to conduct longitudinal studies and the second is to focus on the transfer of "low level" skills. This route has been followed in one part of this thesis, to look at the transfer of low level skills, between two programming languages. The decision to look at students learning low and high level languages sequentially brings two strands of work together. In the context of well formulated conceptual models, it extends the work of Weyer and Cannara (1975), which though interesting, provides rather weak evidence of transfer.

Chapter 3

Chapter 3 provides a theoretical framework for the thesis. The first section begins by considering the three theoretical influences on the study: Piagetian theory, artificial intelligence and educational technology. It goes on to discuss mental models, and clarifies the terminology used in this area. It argues that the use of conceptual models for teaching computing concepts to novices assumes a theory of learning which is broadly Piagetian and that such a Piagetian framework is quite compatible with an artificial intelligence approach. The distinction is made between conceptual and mental models, the role of analogies in conceptual models in a number of domains is considered, and the application of such work to teaching computing concepts is discussed. It is argued that the principles offered by du Boulay, O'Shea and Monk for designing a conceptual model do not address the issue of how the conceptual model is presented. Three further ways of describing the conceptual model are identified and discussed. These are: a state description, a functional description and a procedural description, and examples are given of the

use of each in teaching computing concepts generally. This analysis of how programming is taught indicates that the conceptual models provided often do not give equal weight to these different aspects of the conceptual model, and give least emphasis to the functional description.

Chapter 4

The fourth chapter introduces the four programming environments studied: SOLO, Logo, the assembler used in the PT501 course and DESMOND. First, each language is described to help the reader to follow the examples given in the later data chapters, and then the curriculum and conceptual model associated with each language is analysed and discussed. In each case, the conceptual model is analysed from two perspectives: firstly according to the criteria suggested by du Boulay et al (op. cit.), and secondly focussing on the presentation of the conceptual model and the different descriptions of the conceptual model which were identified in chapter 3.

Chapter 5

Chapter 5 discusses the rationale for the methodologies used in the empirical studies described in chapters 6 to 9, and argues for the use of both quantitative and qualitative methods. One qualitative method used in the PT501 and SOLO studies (chapters 6 and 7) is protocol analysis, and the evidence for the validity of this technique is examined, and recommendations are given for its use. The later studies of DESMOND and Logo employed slightly different methods to the earlier studies of SOLO and PT501. The reasons for this are discussed, and the consequences of using the different methods of investigation, and also the categorisation of the novices' errors and problems.

In the following four chapters the empirical studies undertaken to investigate the

difficulties which novice programmers have are described.

Chapter 6

Chapter six reports on studies of subjects learning to use SOLO. The first part describes preliminary investigations used to evaluate potential methods for carrying out the main investigation. Two different methodologies were investigated: collecting error data at an Open University summer school and collecting think-aloud protocols from individual case studies. The main conclusion was that it was necessary to study a small number of individuals working through SOLO in the laboratory, and a further study was undertaken in order to do this.

The results of this study show that subjects had two main types of difficulties: problems with the domain itself and problems in how they approach learning SOLO. The SOLO problems include difficulties in understanding control structures, writing their own programs, problems caused by the subjects' misinterpretation of the material and erroneous mental models.

Chapter 7

The first part of chapter 7 reports on the first of two studies which investigated the transfer of skills between a high and low level language. In this study, psychology students who had learnt SOLO then went on to learn the PT501 assembler. Because of the high attrition rate, and instruction related problems, it wasn't possible to draw any conclusions about the effects of learning a second language from the study. However, the data suggested that students had particular difficulties in learning to use the PT501 assembler, and these were followed up in a further small scale study. There are two main conclusions. Firstly the students are working at quite a detailed low level. By following the recipe like steps in the

experiment book it is possible for them to get the right answer without a deep understanding of what is going on. There is a mismatch between this structured detailed procedural approach and the evidence discussed in chapter 2 that an important aspect of learning programming is acquiring plans which involves a higher level view of the problem. Secondly the learners' problems can be viewed as instructional problems arising from their interpretations of the text.

Chapter 8

Chapter 8 reports on students' experiences with Logo, as part of the study of the transfer of low level features skills between a high and low level programming language. The first part of the chapter introduces the particular curriculum which subjects followed in learning Open Logo, and gives an evaluation of the Open Logo tutorial manual. The second section of the chapter describes the study and the third summarises the results. In the fourth section, the subjects' difficulties are discussed, and some of these can be traced back to the curriculum. Parts of the curriculum were very hard, and the subjects had not got the pre-requisite knowledge needed for these sections. The interaction of the distance learning hand-holding style and the errors it contained led to severe problems. The subjects also had problems in abstracting salient information from examples given in the text, and partly because of errors in the text, they often failed to get the example programs working - and received error messages which were unhelpful in explaining the problem at an appropriate level.

It is argued that the problems discussed above need to be viewed in the context of Logo's educational philosophy and that there are interactions between the teaching style, the domain and the subjects.

Chapter 9

The second transfer study investigated the transfer of low level feature related skills between high and low level programming languages, where the low level language had benefitted from the experiences of students using the 8049 assembler (course code PT501) and its successor, PT502. The high level language studied was LOGO, and the low level language was incorporated in a small hand held microcomputer called DESMOND. There was no evidence of any transfer effects, and this result is discussed in chapters 8 and 9. Because of this result, chapter 9 reports on students' experiences with DESMOND.

The problems which subjects encountered are categorised into three main groups: domain related, pedagogical and affective. Within the first group, there were two main inter-related problems: identifying and using appropriate programming plans and flow of control, and the relationship between these problems and the instructional design is discussed. Within the second group the problems are pedagogical, and it is suggested that the style of pedagogy and the domain may be in conflict; that the "closed" nature of the curriculum may lead to problems, and that the curriculum requires the students to make unsupported "conceptual leaps" - again leading to problems. Finally affective problems are discussed. The last section discusses the relationship between instructional design and the learners' problems, and suggestions for changes are made.

Chapter 10

Chapter 10 summarises the main conclusions from the various studies, considers the implications and makes suggestions for future research. It provides a critical evaluation of the thesis, and reviews the conclusions of each chapter in terms of their relevance to the field and the implications for future work. The last section suggests future work and extensions to the findings presented.

Chapter 2

RELATED RESEARCH

Contents

2.1 INTRODUCTION

Interest in the acquisition of programming skill has increased dramatically over the past few years: there is now a workshop each year in America devoted to the empirical studies of programmers, and similar workshops are planned in Britain. Although some of this knowledge is influencing how programming is taught, novices still have significant problems in learning programming. One suggestion, mentioned in chapter 1, is to provide novices with conceptual models of the virtual machine that they are dealing with which will serve to anchor the new information that they are receiving to what they already know. However, the provision of such models does not automatically make the problems go away, and so the central concern in this thesis is the problems which novices have when learning to program in environments which provide good conceptual models. Finding out about novices' problems and getting detailed information about the learning process both adds to our knowledge about learning and has implications for how we teach and design curricula.

The literature has been divided into 8 sections, and at the end of each section there is a summary of the main findings and issues and their relevance to this thesis. The studies reviewed here all have a bearing on the problems that novices encounter, but offer quite different approaches. Investigations of programming notation, which are discussed in section 2.2, study the effects of notations on the ease of programming. Looking at experts gives information about skilled behaviour, and by implication what kinds of skills and knowledge novices have yet to acquire. This is discussed in section 2.3. The following section, 2.4, focuses on plans, which are particularly important elements in the acquisition of programming skills. A particular problem for novices is the lack of an appropriate conceptual framework

which can serve to relate new information to existing knowledge. The provision of a conceptual model has been advocated to fill this need, and this is discussed in section 2.5, which also discusses the mental models which novices acquire and use as part of their learning. Section 2.6 looks at cognitive, developmental and educational accounts of novice programming using clinical and observational techniques. The difficulties in learning programming are not dissimilar to the problems faced by novices learning to use other complex systems such as text editors, and so the following section, 2.7, reviews the work in this area. This area is one that has focussed on the use of analogies in teaching and learning: another issue which is clearly of relevance to teaching programming via a conceptual model, as conceptual models often involve the use of analogies.

One of the current gaps in the literature on novice programming is the question of what happens when novices go on to learn subsequent languages. For example is there a 'best' first language or group of languages for facilitating later learning? Section 2.8 discusses the literature that does exist in this area, and finally section 2.9 looks at other work on the languages used in this study.

It is clear from the various sections that interest in how programming skills are acquired is not confined to one field of study; rather the research covers disciplines such as computing, education, psychology, artificial intelligence, cognitive science and ergonomics. Researchers from different fields have somewhat different concerns and approaches; for example cognitive scientists may be concerned with modelling the learner's behaviour, AI researchers with building tutoring systems and educationalists at looking at the benefits of teaching programming in the classroom, but all the studies reviewed are relevant to the problems novices have in learning programming.

2.2 PROGRAMMING NOTATION AND LANGUAGE CONSTRUCTS

Extracting information

How do language notation features affect the difficulties that programmers have, especially in understanding programs? The most extensive work in this area has been carried out by Thomas Green and colleagues at the MRC Social and Applied Psychology Unit at Sheffield University. Green, Sime and Fitter (1980a) argue that in order to understand any text a reader must be able to impose a macrostructure, which reveals the relationships between the different components. In understanding a program, the reader will be following trails, for example, tracing the flow of control, and finding signposts which indicate the macrostructure.

For conditional structures, Green, Sime and Fitter distinguish two different kinds of information that can be extracted. Sequential information is elicited by questions such as "What can it do next after A?" or, "Given that the input is X, what actions will it perform?" Circumstantial questions, on the other hand, ask: "Under what circumstances can A happen?" Circumstantial questions are typically answered by finding A, the first signpost in the example above, and following the trail backwards. For a programmer who is debugging code, or understanding someone else's code, it is vital for such circumstantial questions to be answered. Yet for procedural languages like BASIC it's easier to answer sequential questions than circumstantial questions. For example, in the example segment in figure 2.1 below it is easier to answer the question "What happens after A is printed" i.e. to find out that A is followed by B by working forwards, than to answer the question "Under what circumstances is B printed?" which requires going backwards looking for the jump to 70 and the jump to the jump etc.

```
10 PRINT 'A'
20 GOTO 40
30 .....
40 GOTO 70
50 .....
60 GOTO 99
70 PRINT 'B'
```

Figure 2.1: BASIC type program

Although this schematic example is in BASIC, the ease of sequential tracing over circumstantial is not simply a reflection of the tendency for BASIC programs to be unstructured, but applies to other procedural languages such as Pascal.

Conditional structures

Much of the work in tracing such paths is in following flow of control. Green and colleagues have carried out experiments comparing conditional structures in different languages. Languages like Fortran and Basic use a series of goto commands, whereas languages such as Algol use nested conditionals as shown in fig 2.2 below.

```
        if A then goto L1
        if B then goto L2
        W
L2      V
L1      if C then goto L3
        Z
L3      if D then goto L4
        Y
L4      X
```

BASIC family

```
if A then
    if C then
        if D then x else Y
    else Z
else
    if B then V else W
```

Algol family

Figure 2.2 BASIC vs Algol family (from Green et al., 1980b)

Using microlanguages, the same group compared three different conditional structures: jump-style conditionals, notations with nested conditionals of the usual sort (Nest-BE, i.e. Nest-Begin End) and nested conditionals with added cues to help the reader (Nest-INE, i.e. Nest-If Not End). They found that in the NEST-INE dialect programs were easier to write; fewer errors were made; they were corrected more quickly, and programs written by other people were understood more easily. Green and Arblaster (1980) summarise the earlier papers from the Sheffield group and argue that the problems with these notations arise in extracting circumstantial information, and NEST-INE is best because it combines perceptual cues about structure (for example indenting shows the scope of if) with explicit statement of the negative arm of the predicate. Taken with other findings in this area, however, the results are not totally clear: Green and Arblaster (1980, op. cit) refer to other research which suggests that fully nested hierarchical programs may not be best.

In later experiments the comprehensibility of four miniature programs was compared (Gilmore and Green, 1985). The hypothesis was that comprehensibility depends upon the match between the notational structure and the task being performed, and the results showed that this indeed was the case. Procedural notations, like Pascal, are better suited to answering sequential or forward tracing questions, and declarative notations (for example production systems) are better for backward tracing or circumstantial questions. This is not a surprising result, but it does indicate the pointlessness of searching for an overall "best bet", regardless of the task at hand. One logical next step is to make extracting the harder information easier, using aids to highlight the appropriate structure, and a later study (Green and Cornah, 1985) concentrated on doing just this, by producing a "programmer's torch" as an aid for BASIC programmers.

Principles of notational design

In the paper discussed earlier (Green, Sime and Fitter, 1980a), Green et al also discuss a number of features of programming languages which will affect the ease of use. These are given below:

Discriminability

This is a psychological notion. In programming languages it can be applied to its structures or features: how many are there? How different are they? Good discriminability will help a programmer to identify the macrostructures referred to earlier: "does this test here take part in a conditional or a loop? Easy to tell in a high-level language, hard in a flowchart or an assembly language....Is this loop event-driven or count-driven?" (Green, 1980). Another way of viewing discriminability is to talk about how easily plans or schemata can be seen (see section 2.4)

Generability

Generability concerns how easily a learner can deduce information about other parts of the system from what he or she already knows. In text editing an example would be congruent commands such as "advance/retreat" where the learner has a good chance of guessing the other pair. Similarly the Macintosh computer has high generability in that there is consistency across different applications.

Tractability

Tractability is the ease with which sections of a program can be changed, for example, changing the order of executing subcomponents. Clearly this is linked to structure, and modular, structured languages will allow such changes to be made more easily, but as the authors point out, this only applies to the flow of control, and not the flow of data.

The work on notational features is not directly relevant to work on novices learning programming: as it is geared towards professional programmers, it focusses on the coding and debugging part of the programming process, and applies to the performance, rather than the learning phase, of programming. As will be evident in later sections, the initial problem solving phase where it is necessary to understand and conceptualise the problem, is problematic for novices. I have outlined the design principles briefly however, as one objective of this work is to look at what happens to learners when they are using languages whose designers have followed such principles. Thus, some of these principles will be returned to when discussing the design of the languages used, in chapter 4 .

The relationship between programming and natural language

There are two main questions in this area of research. Given that people successfully use conditionals and follow logical instructions in everyday life, but often have problems in programming, do programming notations map onto the way people specify and solve similar problems using natural language? Secondly, given that our major experience of language is natural language, is natural language used as a model for programming, and is this appropriate?

Miller (1975) compared rules for action in programming languages with those which occur in everyday life, eg. in recipes, instruction manuals etc. His results suggest that where conditionals are used there is a cognitive mismatch between instructions intended for other people and instructions that make a successful computer program. However, research into novices learning Prolog, where they do not use conditionals has shown that problems still occur (Taylor, 1987).

A later study by Miller, (1981) investigated students' natural language solutions of

an information-retrieval problem with the aim of discovering their "natural" procedure specification skills. Subjects were told that another person should be able to follow their descriptions. As in the previous study, Miller found a mismatch between subjects' "natural" approach, and the requirements for producing a computer program, and his conclusion is that: *"programming language style is simply alien to natural specification."*

A follow up study which replicated and extended Miller's work was carried out by Galotti and Ganong, (1985) who argued that there were inadequate controls in Miller's study and also that the subjects did not include obvious information they thought the listener could supply. In particular they looked at the use of control statements and whereas Miller found that control statements were used very infrequently, Galotti and Ganong found they were used more extensively when the listeners were "dumb" Martians, and the task was different.

Bonar (1982) has also researched natural problem solving strategies. The framework for this research is the work on plans by the Yale group (see section 2.4). Bonar suggests that there is a body of knowledge which he calls "natural step-by-step specification knowledge" which strongly influences novice programming behaviour, and he claims that novices use natural language even when supposedly using a programming language. Further work in this area by the same group (Soloway, Bonar and Ehrlich, 1983) has focussed on the relationships between the cognitive strategies used when solving a problem using natural language, and using the programming constructs in Pascal, specifically the WHILE construct. They found that the strategy underlying the correct use of the WHILE construct is not the preferred strategy, which supports their hypothesis that the "natural" spontaneous process is counter to the method which standard Pascal facilitates or supports. As with the studies previously mentioned (e.g. Green,

1980) there is a relationship between task and language notation in that the subjects were able to write correct programs more often when the language facilitated the preferred strategy.

Summary of section 2.2

The design and structure of languages affects their learnability and comprehensibility. The work of Green and colleagues shows that in most procedural languages it is easier to extract sequential information than circumstantial; that particular language structures can make conditionals easier both to write and to follow and that comprehensibility depends on getting a match between the structure of the language, and the task being performed. Improvements in design that result from this work may not help novices, however, as much of the work reviewed on notational features is geared towards professional programmers. Furthermore, this approach is not so concerned with novices' mental processes whilst they are engaged in learning, and the problems that may occur. However, this work has led to principles of design which are of relevance to the design of the languages used in this thesis.

All of the studies on the relationship between natural language and programming languages suggest a mismatch between the two, but as there is a great diversity of natural language style, there is no straightforward solution such as basing programming languages upon natural language.

2.3 EXPERTS AND NOVICES

Looking at experts' programming behaviour may shed some light on what beginner programmers lack that makes programming so hard. Brooks' model of expert programmers is based on a detailed observation of an individual programmer (Brooks, 1977). He categorised programming behaviour into three

stages: understanding, planning and coding. This kind of categorisation is generally accepted and used by other researchers, e.g. Green, Bellamy and Parker, (1987), although it should be emphasised that the stages are not distinct. Within the coding a programmer may return to the planning phases, and from here may recheck his or her understanding of the problem before then going back to coding. Brooks developed an explicit model of the coding process which makes certain assertions about programming behaviour: the most relevant of these is that expert programmers have a large amount of specific knowledge about how to encode particular plan elements. Brooks' model is based on only one programmer but the notion of the expert programmer having a large amount of organised knowledge, probably in small chunks (program schemata) with rules about how to put them together is supported by other research (e.g. Vessey, 1987).

Probably the best known study was conducted by Shneiderman (1976) and is analogous to a classic problem solving study by Chase and Simon (1973) of naturally occuring chess positions. Shneiderman's study indicates that expert programmers "chunk" lines of code in order to remember them, i.e. organise them in ways that are meaningful to them. Together, Shneiderman's study and Brooks' research suggest that expert programmers have a large amount of domain specific knowledge which is highly organised.

Shneiderman and Mayer (1979) developed a model of program comprehension which can be viewed as an extension of the chunking concept. They differentiate between two components in understanding programs: semantic and syntactic knowledge. Semantic knowledge is programming knowledge which includes general concepts that are independent of specific programming langauges, e.g. the concepts of a variable, or of conditional branching. Syntactic knowledge, however, involves the grammar and syntax of particular languages, such as

knowledge of valid character sets or specific conditional statements. Since these rules and conventions may be arbitrary, syntactic knowledge is acquired by rote memorization as opposed to semantic knowledge which is acquired through dealing with programming problems and is stored as meaningful patterns of information.

Jeffries investigated how experts and novices debug Pascal programs (Jeffries 1982). She concludes that novices find debugging very hard and the main difference between novices and experts is not in the strategies they use (or fail to use) but in the novices' lack of understanding of what the program is doing and their difficulty in remembering the programs. They are unable to make use of top down information and are not familiar with the chunks in the way that experts are.

Two other studies which looked at novice-expert differences were carried out by McKeithen et al (1981) and Adelson (1981). McKeithen, Reitman, Reuter and Hirtle studied recall of ALGOL programs by experts and novices and found that the novices could recall less of the programs given than experts, and that this difference disappeared when the lines were scrambled. In a further study of how novices organise key programming knowledge they found no evidence of novices' knowledge being less organised: what differentiated the experts and novices was that the novices' associations was based on common language associations to those concepts whilst experts' organisations was based clearly on programming knowledge. Adelson (op. cit.) used a multitrial free recall task to investigate the underlying organisations of knowledge of her subjects from their ability to recall lines of code. Whilst experts recalled lines by using functional organisation, novices recalled by syntactic class.

Kahney (1982) has criticised the Adelson study for disguising what novices do

know by focussing on experts' knowledge; in other words the Adelson study is not considered to be "culture fair". He argues that the experts could make use of the distinguishing features for each of the programs, whilst for novices understanding the lines of code would require reasoning. But as their task was to recall and not reason, Kahney argues that other forms of organisation may have been available to them but they didn't use them. Soloway's work on novices (Soloway, Ehrlich, Bonar and Greenspan, 1982) shows that novices have a sophisticated ability to work with variables which leads Kahney to suggest:

"Adelson's novices, even if they had not reconstructed the programs as originally written, should be able - using variable names and indices - to indicate which lines belonged to different programs."

Kahney suggests that experts may be able to utilize higher order knowledge in conditions where novices cannot, even though they have it. One of his concerns about Adelson's study is that there is no identifiable point where novices might be gaining competence. However, in the McKeithen study, the novices had begun to acquire some of the chunks of the experts at the end of their first programming course, indicating a general acquisition of knowledge.

Kahney's own research looked at the problem solving strategies of novice programmers. He studied the organisation of programming knowledge in novices and experts and found little difference and argues that it is procedural rather than declarative knowledge which distinguishes experts from novices:

"The differences between experts and novices should ..(be measured) ..in terms of the internal structure of the knowledge itself. ..... That is, two novices may associate recursion and iteration, for instance, but only one may know how and when to use the concept."

More recent work has begun to investigate the nature of the chunks of knowledge that experts have, and has also questioned some aspects of the Shneiderman and

Mayer model. Vessey (1987) investigated the nature of the chunks stored by both experts and novices in long term memory by examining the importance of a match between the programmers' knowledge structures and the knowledge structures contained in the program which they were trying to understand. Her conclusion is that experts have a variety of knowledge structures but do not necessarily have well defined scripts for particular structures: in her study experts did not have well defined scripts for validating records - which was an important component of the program they were given to study.

Various criticisms have been made of Shneiderman and Mayer's model. One issue is whether semantic knowledge is as language independent as is suggested. It is a largely bottom-up model and is concerned with notational aspects only and not with the effects of different tasks on comprehension. Yet the results of Gilmore and Green's study on the comprehensibility of different languages, which was discussed in section 2.2. shows that comprehension is related to the task at hand. Widowski and Eyferth (1986) put forward a similar argument, which is that for a theory of program comprehension, it is not sufficient to focus only on internal knowledge structures of the chunk type (as in the Shneiderman and Mayer model) but it is necessary to also gain insight into the operative features of the language. It may also be necessary to consider additional components of programming knowledge: for example domain specific strategies or heuristics for extracting meaning from complex or unfamiliar programs. They conducted an experiment which supported their hypothesis that the experts' advantage is not restricted to well known or stereotyped structures, but also includes programs which are syntactically correct but not structured in the usual way. They argue that since experts cannot have stored knowledge structures for such programs, there must be other factors involved in their findings that experts handled these programs better than novices, such as domain specific heuristics for extracting and combining

information. They also found that experts had a more flexible reading strategy: they changed their strategy according to whether the programs were structured in well known ways or not.

Summary of section 2.3

All these studies support the notion that expert programming skill can be viewed as a large amount of organised programming knowledge and rules for putting it together, and knowing when to use which bits. The evidence on how novices' and experts' knowledge differs is less clear, but there is some consensus that novices and experts categorise what programming knowledge they do have differently.

2.4 PLANS

In section 2.3 it was seen that programming skill involves a good deal of specialised knowledge, organised as plans. This section looks at the research on that plan knowledge which stems from the work on expert-novice differences which was discussed in the last section.

Soloway, Ehrlich, Bonar and Greenspan (1982) have built on the work of Shneiderman (1976) and Adelson (1981) to develop a theory of expert plan knowledge. According to this theory plans represent stereotypical actions in a program: they are high level structures which serve to bring together related pieces of information. For example a "variable plan" would summarise the various kinds of knowledge regarding the use of that variable in the program: its role in the program (e.g. counter variable, running total variable), its initialisation and update, and the guard which may protect a variable. Another example is a "running total loop plan" which is a control-flow plan which keeps the total of a list

of numbers. Soloway et al (op. cit.) classified the plans as belonging to one of three categories. The first two are strategic and tactical plans which are both language independent, and the third includes implementation plans which are language specific. Strategic plans specify a global strategy used in an algorithm. For example, the "Read/Process" strategy specifies that the actions "read in a value, then process it" are nested in a repeat loop, as in the example in figure 2.3 below.

```
repeat
  Read (Next);
  Sum := Sum + Next
  Count := Count + 1
```

Figure 2.3 : The Strategic Read/Process Plan

In the plan shown in figure 2.3, the variable "Next" is first read, and then it is processed in the statement "Sum := Sum + Next". Tactical plans specify a local strategy for solving a problem. For example, the "Counter Controlled Running Total Loop Plan" describes how a sum can be accumulated. It specifies an algorithm for solving a specific problem. Implementation plans specify language dependent techniques for realising tactical and strategic Plans. For example the "*for* loop plan" is a technique for implementing the "Counter controlled Running total loop plan" in Pascal.

This plan theory has been used as the basis for classifying bugs, (Johnson, Draper and Soloway, 1983) where buggy plans are represented as deviations from the correct plan. However, the given dimensions and categories do not unambiguously classify bugs, and as Gilmore points out (Gilmore, 1986), closely related errors are awarded quite different categories, because it is a behavioural rather than a cognitive classification system. Soloway and Ehrlich (1984) also discuss another

kind of expert knowledge which is related to plans: the rules of "programming discourse". These specify conventions and are seen as having a similar role to the rules of discourse in natural language. For example, it is expected that the name of a variable agrees with its function. Violating such rules of discourse results in programs becoming"unplanlike", and on such programs, experts' performance is reduced to that of novices, both in using cloze procedures (i.e. filling in blank lines) and on recall tasks.

Rist's research (Rist, 1986) is also on plans, but emphasises the interaction between the plans and the program's goals. The program is viewed as a goal directed artefact that is constructed by selecting plans that satisfy the program's goal. The plans organise the code into well defined fragments. The basic programming plans described by Soloway et al (op. cit.) are small programming fragments that achieve a single well defined goal such as to count or sum a series of numbers, but these basic plans by themselves do not describe a program; they must be accompanied by a plan and goal description at a higher level of analysis. A program can therefore be considered as a plan tree that relates program plans to dominant goals where the program goals represents the highest node and below this are standard global plans, such as input, process and output. At each level of analysis the goals are split into plans and the process continues until the plans at the lowest level can be translated into code.

Rist investigated the plan structure used by both novices and experts in a task where they grouped together lines of code in a number of programs. The experiment established the use of plans by both novice and expert programmers. In understanding the code in a program, the experts defined code groups on the basis of program plans that the code implemented. The novices, however, used a mixture of syntactic, control and plan based groups, reverting to more syntactic groupings when the programs became more complex. There was a trend towards

increasing plan use with expertise and decreasing plan use with difficulty. Plan knowledge was divided into three main types: global, focal and basic.

Global plans (Gplans) were the first plans taught and were used by novices and experts similarly. These are used at the highest levels of program description. For example the basic structure of a program is to *input* a set of data, *process* the data or calculate a result and *output* the result. These global plan descriptions define simple programs completely, but not more complex programs. Secondly *focal* plan descriptions appeared in both novice and expert trees. Focal plans appear deep inside the program and the plan tree. In more complex programs the focal segment may be buried at the bottom level of the plan tree in the most intricate code. For example, in a sort program the test and order process that compares two elements is focal to the sort, and the focal line can be viewed as the driving force of the plan. Finally, *basic* plans define the detailed structure of the program. These are the plans described by Soloway et al (op. cit.) such as count and sum and are shown in figure 2.4.

Role		**Plans**	
	Count	Sum	Average
Init	count := 0	sum := 0	av := 0
Process	count := count + 1	sum := sum + 1	if count < > 0 then av := sum/count
Use	calc average	calc average	writelm (av)

Figure 2.4: Plans: count, sum and average (from Rist, 1986)

Rist's approach here is to view program construction as a kind of problem solving: "where in the first version of a new program, a new problem is written by a procedure of goal back-chaining" (Rist, 1986). That is, in order to solve the overall goal, sub-goals are created which in turn require the solution of further

subgoals. The initial definition of plans comes from the goal chaining technique.

An example of the interaction of plans with the goal chain can be seen in the *average* plan which makes use of the count and sum plans shown in figure 2.4. The products of the count and sum plans are used to calculate the average. The process segments of these plans are contained in the read loop which is itself a complex plan which can be sub-divided further. These plans describe the plan structure of the average program, and they may be tied together by constructing a goal chain to which the plans are attached. The goal chain begins at the task of the program which is to output the average of a set of numbers. This creates a goal to calculate the average, the focus (process) of the output goal. The calculation goal creates the count and sum goals which in turn require numbers as input. This creates the need for an input goal and finally a loop to repeat over the numbers.

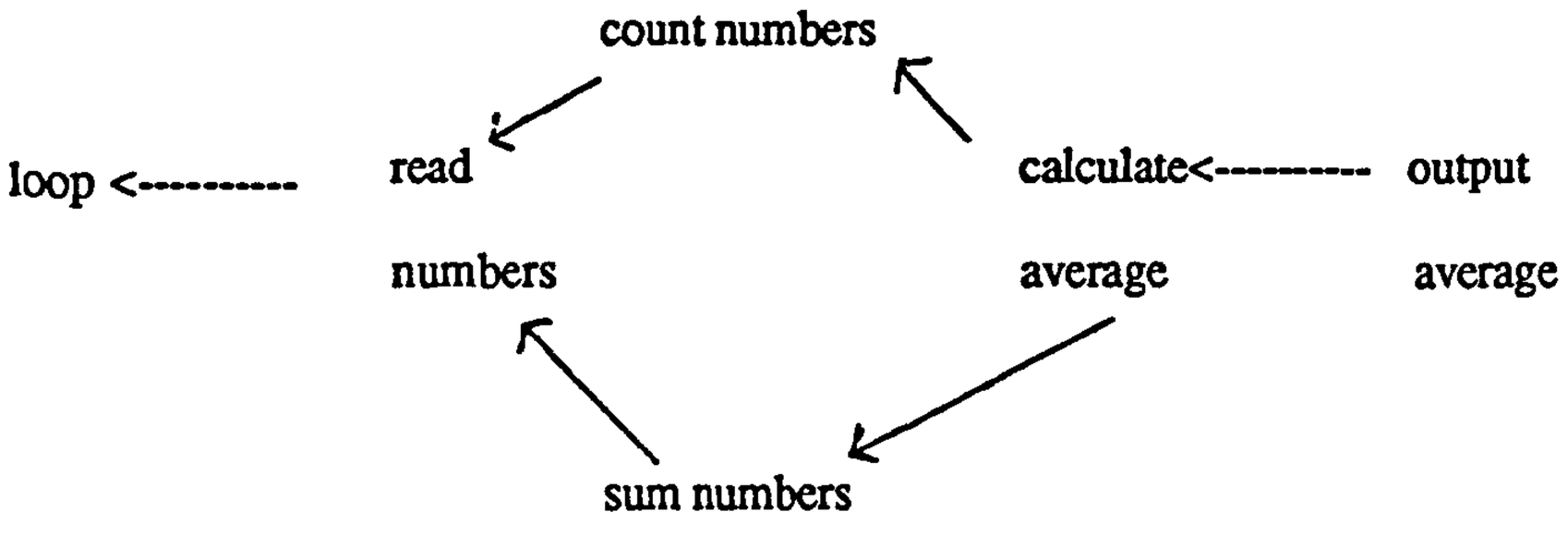


Figure 2.5: Goal chain for finding the average (from Rist, 1986)

Weidenbeck (1986) draws on work on plans in her research on comprehension, which unlike the Shneiderman and Mayer model discussed in section 2.3 is a top-down model. Her model of comprehension is a 3 part process where the programmer generates a hypothesis (of program structure) to direct a search, the

search then samples the program data, and the program data modifies the hypothesis. She has investigated the use of beacons - which are a typical indicator of a structure or operation, and are first mentioned by Brooks (1977, op. cit.). They consist of information which is expected and stereotypical. A memory and recall experiment supported the existence and the use of beacons in program comprehension. She concludes that these key features in programming are the focal points for programmers as they study and comprehend a program, in that they are a very characteristic part of the code. They help programmers to work out the general high level format but do not contribute to detailed understanding of the code, which may be needed for debugging or modification.

The most detailed investigation of the psychological nature of plans has been carried out by Gilmore (1986) who has investigated the status of plans. He points out that although the notion of plans has been widely accepted, and their use by experts demonstrated (e.g. Rist, 1985) they have not been clearly defined. They are not necessarily algorithms (although most of the Yale examples are algorithmic) but are code fragments which when interleaved provide the whole program. Expert programmers tease them apart when reading a program and plait them together when writing. Their role in developing expertise is not straightforward. Rist's work, which has already been discussed, (Rist, 1985, 1986) demonstrates that novices can benefit from plans as much as experts - and use plans, and suggests that the growth of expertise replicates the movement from surface (or syntactic) to deep plan structures. Kahney (op. cit.) also found evidence of the use of plans by novices. Gilmore's main question is whether plans are fundamental to programs: do they represent the deep structure of a problem or are they just another (albeit very important) information structure within the code? Another question is whether they apply to languages other than Pascal, the language studied by the Yale group. Gilmore found no evidence that BASIC programmers understand and use the same plan structures as expert Pascal

programmers. There are a number of possible explanations for this result. For example, plans may not be applicable to Basic in the same way as to Pascal, Basic programmers may differ from Pascal programmers (i.e. be drawn from a different population), or the teaching techniques used may be different. Davies (1988) replicated Gilmore's study using Basic programmers who had been taught in a structured way and found that highlighting plans did help their comprehension.

Gilmore concludes that plans do not represent the deep structure of the problem but are a non-syntactic view of the code with which experts are proficient. This does not resolve the generalisability question, but there is evidence that Lisp programmers use similar plan structures (Soloway 1985). Hasemer (1983) has described the SOLO segments which are used by his debugging aid AURAC which suggests both that they may be generalised beyond Pascal, and even if they do not represent the deep structure of the problem they are clearly very important.

Summary of section 2.4

As we saw in section 2.3 there is some agreement that novices and experts categorise their programming knowledge in different ways. For experts, plans are clearly one important way of organising such knowledge, and have been used as the basis of bug classifications and are used in comprehension. Most of the work on plans, however, has been on Pascal, which raises the question of how generaliseable they are. Gilmore has made the most comprehensive study of plans and found no evidence of their use by BASIC programmers, but Davies's replication study indicates that plans can be used by BASIC programmers, if they have been taught in a structured way. Additionally there is weaker evidence of their use in other languages, for example LISP.

This work concentrates on the skills that novices haven't got, and so its main contribution, therefore, is to increase our understanding of the nature of programming skill, to emphasise the importance of plans for experts, and to outline the limitations of plans as forming the fundamental structure of programs. However, there is also some evidence of their use by novices and some of the errors made by the novices studied in this thesis are analysed in terms of plans.

2.5 CONCEPTUAL AND MENTAL MODELS

What are conceptual models, and what is the relationship between them and notional machines, and abstract machines? This section begins by discussing the terminology, and clarifying the terminology that will be used in this thesis before moving on to discuss the literature. Young (1981) has stated that *"system designers and applied psychologists are increasingly coming to believe that people deal with complex interactive devices by making use of a conceptual model of the device, and that the fostering of this model is an important consideration for the designer. The notion of a user's "conceptual model" is a rather hazy one, but central to it is the assumption that the user will adopt some more or less definite representation or metaphor which guides his actions and helps him interpret the device's behaviour. Such a model, when appropriate, can be helpful, or perhaps even necessary for dealing with the device, but when inappropriate or inadequate can lead to misconceptions and errors."*. The notion has not become much clearer since this time.

This thesis is concerned with the mental models which learners themselves develop and use in learning a domain. This term will be used to refer to what Young called a user's conceptual model: i.e. the user's representation of the notional machine.

Norman (1983) has offered the following definitions of various models and how they relate to the system being taught. The target system is by definition, the system the person is learning to use. The conceptual model is invented by the designers of the target system and/or educators to provide a representation of the target system. This representation is appropriate in the sense of being accurate, consistent and complete, and it is presented to the student by means of a system image which appears to be almost identical to the notional machine of du Boulay, O'Shea and Monk (1981) which will be discussed shortly. The mental model is the mental representation built by the learner herself. Norman comments that these models are not the tidy, complete and consistent models which might be hoped for, but are often incomplete and unstable: they *"contain only partial descriptions of operations and huge areas of uncertainties"*. Norman (1983)

Norman discusses mental models in various computer-related domains (such as calculators) and advocates basing the design of new systems around a conceptual model and creating a system image that is consistent, cohesive and intelligent. These recommendations are rather similar to those of du Boulay and O'Shea's. In such a situation, Norman argues, assuming the system image is consistent with the underlying conceptual model, the learner's mental model will also be consistent. This study is concerned in part with this claim, as it investigates the problems students have when using a system with what Norman calls a system image. There seems to be no advantage in making the distinction (as Norman does) between the conceptual model and the way it is presented. In this study, therefore, a conceptual model will be defined as a representation of the target system. The curriculum designers will be using a range of methods to present this model to students. It will not necessarily be a completely accurate representation of the target system, as an analogy may only partly correspond, but nevertheless be useful.

The rest of this section will discuss the literature on conceptual and mental models. Du Boulay has recently discussed in general terms some of the problems novices have in managing the notional machine (1986). One widely accepted solution is to provide some form of conceptual model for novices (du Boulay, O'Shea and Monk (1981) Mayer (1979), Norman (1983)). The idea is to present the workings of the notional machine (representing the system which novices are to learn) by references to mechanisms with which novices are familiar, and by giving novices as clear a 'story' about the workings of the machine as is possible.

Du Boulay, O'Shea and Monk's glass box

Du Boulay, O'Shea and Monk (1981) argue that a major problem in teaching novices programming is describing the machine which the novice is learning to use at the right level - as the novice usually does not know what the machine can be instructed to do or how it manages to do it. They suggest overcoming this problem by basing the teaching on a "notional machine". This is an idealised model of the computer implied by the constructs of the programming language: i.e., it is not related to the hardware but is language dependent: thus a BASIC machine is different from a LISP or Pascal machine. The notional machine provides a descriptive model which the student can use to plan programs and to interpret the response from the machine. They advocate two principles upon which to base the notional machine: conceptual simplicity and visibility.

Simplicity

Three types of simplicity are discussed. Functional simplicity means having a set of instructions and commands which is small enough to be readily learnt by a novice. Logical simplicity implies that the language is suited to the task in hand. Syntactic simplicity:*"is achieved by ensuring that the rules for writing instructions are uniform, with few special cases*

to remember and have well chosen names." (du Boulay et al., op. cit.).

In practice, it is not always easy to identify these characteristics, or to distinguish between the three types. Functional simplicity is clear enough, but the issue of how one decides whether the language is suited to the job that the novice wants doing is a bit more tricky (e.g. one could imagine heated debates about the suitability of Prolog or BASIC for various tasks that novices might want to do). The problem with syntactic simplicity is the name. This is closer to what might be called consistency, and is similar to Green's 'generability'. Consistency would also include the notion that all aspects of the teaching materials should be consistent. Du Boulay et al see this element of consistency as being part of visibility, as it is to do with the commentary that is provided: *"The commentary, whether pictorial or written, should be at a level of detail appropriate to the novices's task and to his level of understanding of the underlying concepts. Its terminology and diagrams should be properly matched to the other explanations that are provided e.g. in teaching materials."*

Visibility

Wherever possible, methods should be provided for the learner to see the workings of the notional machine in action, for instance the effects of commands. One way of doing this is by means of a commentary: a 'glass box' through which novices can see the workings of the machine. Not surprisingly most of the examples of visibility which are given concern highlighting flow of control. Help for novices to decide what state the machine is in is also mentioned, also the importance of choosing names that do not conflict with novices' previous knowledge.

Du Boulay, O'Shea and Monk give three examples of notional machines which

embody the principles of both simplicity and visibility One notional machine for a microcomputer (which will be referred to as PT501, the code number of the course for which it was designed) is essentially provided by the machine itself - it is its own model and has been designed so that its workings, at the apropriate level, are visible (Open University, 1979). The other two notional machines are for the programming languages SOLO (Eisenstadt, 1978) and Elogo (McArthur, 1974). In SOLO, some of the workings of the notional machine are in the language itself, for instance SOLO automatically shows the learner the updated state of the database after an item has been inserted or deleted, and others are described in the manual, sometimes by using analogies. In using ELogo, novices progress from using a turtle and button box to using a teletype; but are still protected from parts of the system which they don't need to know about.

Although the principles themselves seem to be very laudable, it is sometimes hard to distinguish between the two categories given, and so it is not clear whether this categorisation holds up. For example, the issue of choosing names was discussed under visibility, but it seems it could as easily be a question of simplicity, or consistency if such a category existed. Another problem is that there may not be a clear correspondence between what appears to be good design, according the principle of simplicity, and how easy such systems are to use in practice. Whilst other researchers (e.g. Mayer, 1975) have carried out empirical work demonstrating the value of conceptual models as such, du Boulay and O'Shea's recommendations have not been empirically tested. However, the three examples of systems that they give have all been used successfully, so this could be seen as a 'naturally existing' experiment, the results of which are clearest for SOLO and Logo as SOLO students complete assignments based on their SOLO work, and there have been studies of the use of ELogo. The data reported in this thesis, however, shows that there are learnability and usability problems in the three languages mentioned.

Mayer's work on novice programmers

In discussing the necessary conditions for meaningful learning to occur Mayer (1979a) uses a model of learning in which the learner is in contact with the new material, brings it into working memory and then searches long term memory for appropriate knowledge to which the new information can be linked. Such knowledge must then be transferred to working memory so that it can be combined with new information in working memory. A similar model underlies much of the work in this area: in particular work on learning by analogy, such as that discussed in section 2.8, but it is often not made explicit.

Assuming such a framework, Mayer (1975) proposed a concrete model for teaching a BASIC-like language. This provides analogies for four functional units of the computer, and can either be presented as a diagram or as a board using actual parts.

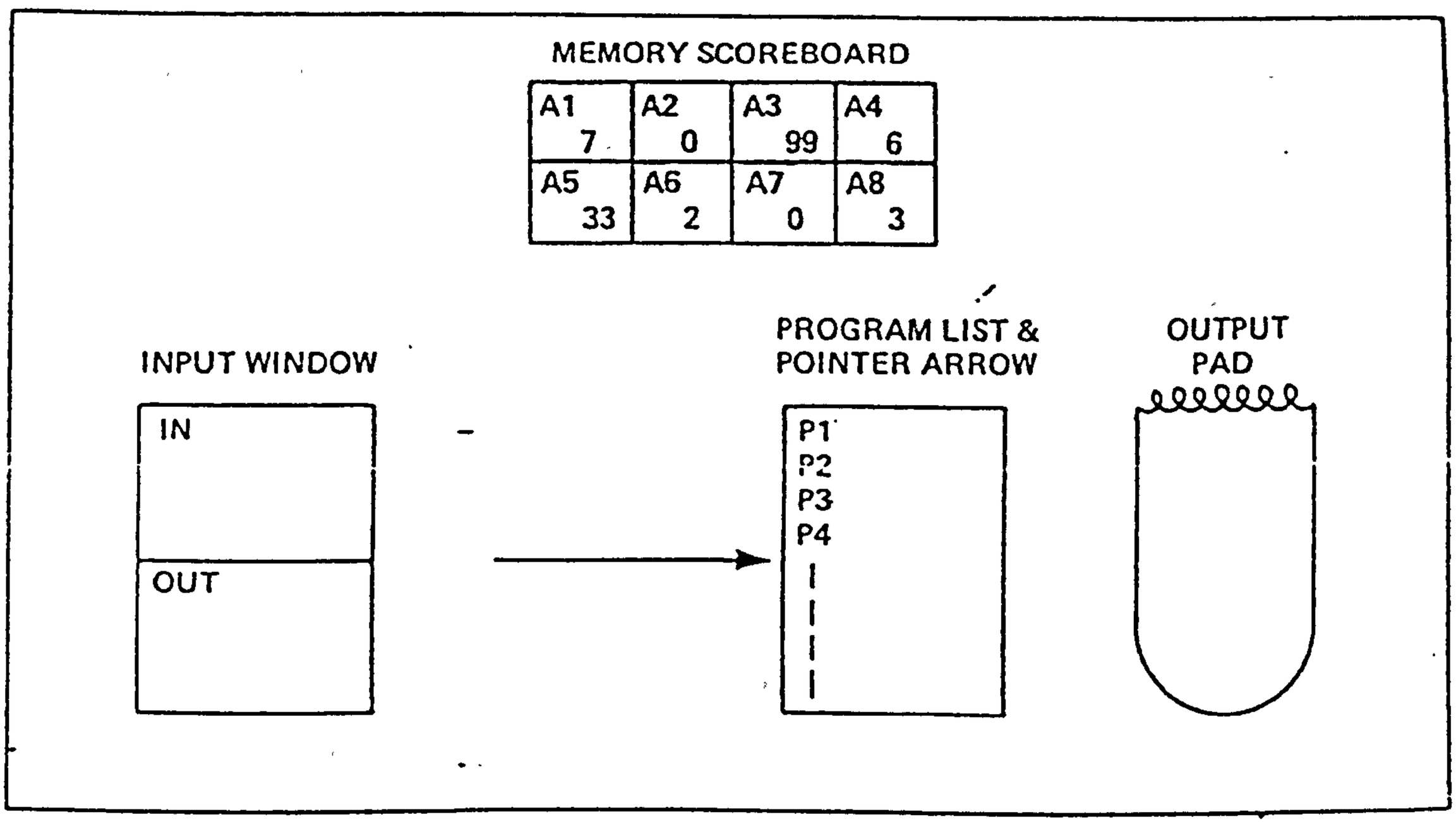


Figure 2.6: Concrete model of the computer for a BASIC -like language (from Mayer, 1975)

Mayer investigated the helpfulness of this model in a study (Mayer 1975, op.cit) where subjects read a short manual describing BASIC-like statements and then took a test which included both generating and interpreting code. One group was given the model before reading the manual and the other was not. On problems requiring a moderate amount of transfer the model group performed significantly better than the other group. This result was repeated in a further study (Mayer 1976) where all subjects had the model but half the subjects had the model before reading the manual, and half had the model afterwards. On a different set of recall tests, the group given the model before reading the manual was better at recall of conceptual information. In both studies the results were strongest for low ability subjects, and a further study indicated that such results were also more pronounced when material was poorly organised (Mayer, 1978).

In a follow up study, students learned a file management language with and without a concrete model (Mayer, 1980). In this study Mayer also investigated the effects of elaboration: encouraging learners to explain information in their own words and relate the material to other concepts or ideas. The group who were encouraged to elaborate tended to emphasise recall of conceptual information whilst the other group recalled technical and 'format' information.

Mayer has also developed "transactional analysis" (Mayer, 1979b) as a framework for describing a learner's conceptual knowledge of BASIC, and these transactions can also be exploited for teaching novices: *"The transactional analysis approach involves taking a BASIC statement such as INPUT A or LET B + A + 1 or PRINT B and trying to understand its microstructure and macrostructure. Microstructure refers to the conceptual parts or units that are referred to by the statement; macrostructure refers to the larger structure into which the statement*

fits within a program or program segment." (Mayer, 1987). A transaction can be thought of as a unit of programming knowledge and it can be broken down into three parts: the operation - such as MOVE, FIND, CREATE etc.; the objects which is acted upon - such as number, program, pointer, etc.; and the location of the operation - such as memory space, output screen, keyboard, file, etc. For example, the transactions for PRINT B are:

1. Find the number in the memory address indicated. (FIND, Number, Memory)
2. Write that number on the next available space on the output screen. (CREATE, Number, Screen)
3. Go to the next statement. (MOVE, Pointer, Program)
4. Do what the statement says. (ALLOW, Command, Program)

The importance of this instructional level is in providing a means of explaining what is going on inside the computer when a particular statement is executed and of relating the new technical language to general operations, locations and objects that he or she is familiar with. The results of a study by Bayman (1983) support the idea that direct instruction in underlying conceptions (by using transactional analysis) can be effective and that such training can enhance students' performance in solving programming problems.

Analogies and metaphors

Carroll and Thomas (1982) have been particularly concerned with the use of metaphors for introducing command languages and text editing systems to novices, and they suggest some principles for deciding whether a particular metaphor may be a good one, or for deciding between metaphors. They emphasise the notion of congruence: that a metaphor must have a suitable relation to what it is intended to represent, and give the example of a typewriter metaphor being used to introduce text-editors. Both have a similar function, with similarly laid out keyboards;

however, the cursor of the text-editor does not have an equivalent in the typewriter, and being constantly told to save the document does not fit in with the typewriter metaphor either. The point is, that no metaphor fits perfectly, but the more aspects of the system that can be covered by a single metaphor the better.

Work by Bott (1979), also on learning to use text-editors, suggests that metaphors which match very closely may be less helpful than those which match a little less closely when the metaphor is generated by the learner and is therefore not known to the teacher. This is because the learner's experience is that the two systems are the same, and so he or she develops mental models on this basis, which are hard to change, as they fit 99% of the time, yet the other 1% when they **don't** fit, may, of course, be crucial. Bott's work is discussed further in section 2.7. The idea of using analogies as part of the conceptual model used for teaching novices is taken up in chapter 3, which also considers the role of analogies in producing theories which attempt to account for complex learning.

Few researchers in this area have attempted to understand the sources of novices' difficulties when using computer text-editors. Allwood and Elliasson (1987) analysed the causes of novices' errors and inefficient commands in a text-editing task. Their results show that deficient analogical reasoning may have contributed to a large proportion of the errors. When only errors were considered, analogies from the currently used program led to more errors than analogies based on knowledge about typewriters. However, the study also investigated the use of inefficient commands, and when they were included in the analysis the two sources of analogies were of equal importance.

Summary of section 2.5

This work is particularly relevant because the novices who are being studied are learning programming with the help of conceptual models. Although du Boulay,

O'Shea and Monk offer some criteria for designing instruction around a conceptual model, there has been no empirical investigation, hitherto, of their criteria. Mayer's work has demonstrated the helpfulness of conceptual models in general, although they are most helpful for less able learners or cases where the instructional material is poorly organised. His work also indicates the usefulness of other techniques such as elaboration. Finally, the work in this area is also of interest because of its focus on instruction.

2.6 CHILDREN, PROBLEM SOLVING AND FIELD STUDIES OF NOVICES

The research discussed so far leaves a great deal of novice behaviour unaccounted for, much of which is behaviour "in the field": what happens in typical programming classes. This section looks at two kinds of studies: research on the kinds of problems (and achievements) experienced by learners as part of their normal school curriculum, and investigations of the cognitive benefits which may result from learning to program. These studies give cognitive, developmental and educational accounts of novice programming using clinical or observational techniques. This section is the only one which discusses research on children learning programming. There is no evidence that the problems which older children encounter are any different from those of adults: on the contrary, the problems appear to be very similar and so the results from these studies are assumed to be generaliseable to adults.

Patterns of learning

Perkins, Hancock, Hobbs, Martin and Simmons (1986) researched the different patterns of learning which children adopted when using BASIC or Logo as part of

their normal school curriculum and identified a number of strategies for coping with problems. There are two opposing characteristics of disengagement from the problem, and children exhibiting these characteristics are labelled as "stoppers" and "movers". Stoppers simply give up when they hit a problem - even though they may have the skills to solve it. The authors relate this to powerful affective factors, for example an evaluative view of bugs which views bugs as indicating the students' failure. At the other end of the spectrum are movers who constantly try one idea after another, writing code and changing it and never stopping long enough to appear stuck. While movers clearly have some chance of solving a problem, extreme movers are exhibiting the same disengagement as stoppers - in that their actions are unreflective and impulsive and so they are only likely to solve their problem by trial and error.

The children also varied in the extent to which they engaged in close tracking of code, i.e. reading the code carefully to determine what it does. In effect this is mental execution, and it requires understanding of the primitives of the language and the rules of flow of control. Students commonly failed to engage in close tracking.

Tinkering is the attempt to solve a programming problem by a series of small repairs in the code It is related to the theme of stoppers, movers and trackers, as effective tracking depends on close tinkering and being systematic. If however there is no diagnosis of the problem - and no close tracking, and if the tinkering is unsystematic, it can be disastrous. The final phenomenon is breaking the problem down, the success of which is affected by several factors, not least of which is the knowledge that this is a helpful way to approach a problem. This kind of knowledge is a weak general problem solving strategy, the role of which is discusssed in a different paper by the same group.

Perkins and Martin (1986) looked at the success of a group of 20 high school students taking a one year course in BASIC. Their task was to tackle a series of problems of increasing complexity: the students chose the first problem to tackle and continued until they ran out of time. The experimenter worked with children individually and provided help only if it was needed in order to solve the problem. The help ranged from general strategic prompts (which were tried first), through hints to giving specific advice.

Their main finding is that the students they studied were hampered by *fragile knowledge*, i.e. having some fragments of knowledge "without being able to marshall enough knowledge with sufficient precision to carry a problem through to a clean solution". Such fragile knowledge is exacerbated by a shortfall in elementary problem solving knowledge. Four types of fragile knowledge are discussed: missing knowledge, inert knowledge, misplaced knowledge and conglomerated knowledge. The successful prompts given to students can be seen as high level strategic questions that students might ask themselves. Such questions were to do with aspects of problem solving such as formulating goals, generating a solution, evaluating the solution etc. The authors discuss three general prescriptions for teaching programming to help overcome these problems: teaching so as to reduce fragile knowledge by highlighting the functional roles of commands in the generality; preserving the exploratory use of the language and encouraging the use of problem solving strategies.

Another field study is Linn's study of different levels of programming skills in middle school programming classes (Linn, 1985). Linn outlines a chain of cognitive accomplishments which represents a model of what can be learned from programming courses. The first link of the chain is learning language features, followed by learning to design programs to solve problems (which includes

developing a repertoire of plans and procedural skills). The next link is learning problem solving skills applicable to other formal systems - which includes developing a repertoire of generalized plans suitable for adaptation to new formal systems. The final link is explicitly identifying generalized procedural skills for planning, testing and reformulating problems in a variety of formal systems. The chain of accomplishments is partly based on research reported in the literature as well as Linn's own observations and also draws on successful texts. This paper focusses on the relative contribution to successful programming of general problem solving skills and knowledge (or lack of it) of programming constructs:*"One can speak roughly of a continuum between the low-level knowledge of the particular commands a language offers and general tactics of problem solving "* (Linn 1985). With such a continuum in mind, is the shortfall principally in low level knowledge or high level strategic repertoire?

The rest of Linn's paper discusses the progress of 600 middle school students along this chain of accomplishments, using BASIC, and the fact that the students did not progress very far along the chain. Classrooms were categorized as either typical or exemplary. The 15 schools selected as being typical ran courses running at least 12 weeks, had at least 8 computers and teachers who had participated in in-service training workshops. At the end of the instruction period, students at typical sites remained at the beginning of the chain of accomplishments. Three classes offering exemplary instruction were also located and selected because here the students were explicitly taught how to design programs. They differed from typical schools in that the teachers were more experienced and the students were on average somewhat higher in general ability. In these classes students were able to design programs to solve problems, i.e. they had reached the design link of the chain of cognitive accomplishments.

To see if any students had acquired the general problem-solving skills on the chain of cognitive accomplishments, the most talented students were sought and studied intensively. Although the testing was rather informal, observation revealed that these students used generalised procedural skills. One interesting (but perhaps not surprising) finding is the tremendous range of student performances after an introductory course. Average performance in the final test ranged from 17% at one typical school to 86% at an exemplary school. Measurement of ability suggested that for learning language features at typical schools, general ability was important, but for learning design at exemplary schools it was not so important. Again, exemplary and typical schools differed in the effects of access: in exemplary schools home access to computers was not directly related to performance but in typical schools it was. Finally boys were more likely to have previous experience than girls and in typical schools this related to performance, but again not in exemplary schools. Although girls formed only 37% of the programming students they formed 60% of those in the talented group.

Cognitive effects of learning to program

The relationship between programming and problem solving and the cognitive effects of learning programming has also been studied by researchers at Bank Street College, New York. The best overview of this research and related issues is given by Pea and Kurland (1984). This paper examines two opposing beliefs about the teaching of programming. The first is that learning programming is an accumulation of facts, i.e. learning to use a programming language consists of learning its notational features. The second belief is that children acquire powerful problem solving skills through programming. The perspective taken is described as developmental cognitive science: *"The synthesis of developmental cognitive science focusses on diagnosing the mental models children and adults bring to understanding computer programming since these models of processes*

serve as a basis of understanding of transformation to their systems of knowledge as they learn"

The paper reviews the literature from this perspective with respect to the two beliefs mentioned previously. The first is presumably included so as to include two "extreme" positions: it is unlikely that anyone who has given some thought to the issue would hold this belief. However, programming is often taught as though programming is mainly a matter of learning notation and the achievements of the children in Linn's study who only reached the features link of the chain may reflect such an approach to teaching. The second claim is also given in its extreme form, and although there have been claims of this kind made quite frequently this paper is one of the few that examines such claims thoroughly. It argues that the expected benefits of acquiring powerful problem solving skills claimed by Feurzeig (as cited by Pea and Kurland, 1984) are unlikely to happen for two main reasons, Firstly, transfer itself is unlikely as it is notoriously difficult for people to recognise problem isomorphs, and secondly recent research fails to support the notion of abstract problem solving skill, i.e. non domain specific skill. The paper also analyses the nature of programming skill. Skilled programming can be viewed as a set of activities constituting the phases of problem solving (what the programmer does) and what the programmer knows. Four levels of programming skill are distinguished: - 1) program user, 2) code generator, 3) program generator and 4) software developer. The authors conclude that there is little evidence for the cognitive effects of transfer but point out that studies have looked at high level outcomes but given students very little programming experience. There is therefore a mismatch of treatment and expected outcomes. As Pea and Kurland point out in their conclusions there are large gaps between what is meant by learning to program in the computer science literature and what learning to program means to educators in this field. It is unreasonable therefore to expect powerful outcomes from modest exposure - and as Linn points out, from

modest teaching. Two routes are suggested, one is that studies focus on "low level" transfer : skills of levels 1 and 2 in the authors' hierarchy. The route which may make it possible to address the transfer of skills at a higher level will require longitudinal studies where the learner's exposure is much higher.

A later study by Kurland, Pea, Clement and Mawby (1986) investigated the relationship between thinking skills and programming and also the development of programming skills acquired by high school students. Three groups were involved; one which had no programming experience, one which had done an introductory programming course and a third experimental group which during 36 weeks spent 40 minutes 5 days a week studying 6 languages altogether. The teacher of the experimental group was highly qualified and considered by the researchers to be excellent. One of the languages studied was Logo, and here the curriculum was designed by the researchers to help the students develop a rich mental model and to focus on control structure.

A comprehensive array of tests was administered to each group. No differences were found between the groups on pre-test scores and for most tests there were no differences on the post-tests scores. The exceptions were the algorithm design and analysis test where the programming students were more likely to use 3 out of 4 possible programming structures (loop, conditional test and counter) than the other two groups. Even in these favourable conditions of a relatively intensive curriculum and exemplary instruction all the measures of programming skills showed that most students had gained only a modest understanding. In their Logo proficiency they exhibited a "somewhat confused overall understanding". The authors conclude that such failure to transfer is not because programming skill is disconnected from other skills: the reasoning and maths measures correlated with programming performance indicating that programming taps a number of specific

complex cognitive skills. The programming students did perform significantly better, however, on the algorithm design and analysis test. The authors suggest that where they could see the relationship between their programming knowledge and the task they were being asked to do, they were able to transfer their skill. In the case of the successful task, the knowledge to be transferred included specific programming concepts such as "counter" and conditional stop rule as well as the cognitive operations used in programming (e.g. procedural reasoning) and the task bore obvious similarities to a programming task, so the students recognised the conditions for applying some of their programming concepts to the task.

One problem with this study is that students did not gain much proficiency, so it could be argued that there is little skill to transfer. The study does not indicate, therefore, whether a deeper level of expertise makes a difference as the students did not attain a very high level of expertise: and so the benefits that may come with the level of skill acquired by professional programmers are still speculative. However, the successful transfer of skills may only involve tasks that are closely related.

Summary of section 2.6

This research is of particular interest as it is concerned with how children learn on typical programming courses rather than in laboratory settings. Such courses are closer to the conditions experienced by the learners who were subjects for this thesis. Unlike the work reported in this thesis, however, the curricula were not designed around conceptual models. The studies reviewed report on quite different findings, ranging from the styles of learning (and not learning) to the particular skills learned (or not learned). The pervasive result throughout these studies is the lack of proficiency gained by most students learning programming on typical courses. The other main finding is the lack of support for the idea of

programming as a vehicle for developing powerful problem solving skills. Pea and Kurland relate this to the small amount of exposure to programming on most programming courses. Given this, they suggest two routes forward , one of which is to focus on the transfer of "low level" skills. In part of this thesis this route has been followed, and empirical studies were conducted which looked at the transfer of low level skills between two programming languages. What Kurland refers to as a developmental cognitive science perspective is similar to the perspective taken in this thesis, although the focus is on adults rather than children

2.7 LEARNING TO USE COMPLEX DEVICES

As this research is concerned with the problems novices have at the beginning of the learning process, much of the research in related domains, such as text-processing, is relevant. Studies on text-processing are often situations where learners are using self-study manuals, and given that this thesis is devoted to distance learning, this is another reason for interest in this area.

One of the most active groups in this area is at the IBM Watson Research Centre in the U.S.A. Mack, Lewis and Carroll (1982) discuss a study where the subjects (who were office temporaries) learnt one of two text processing systems. They used self study manuals which they studied for about 12 hours over four half days, and in the testing phase they were asked to type and revise a letter. The task took about an hour, and they were asked to think-aloud while they worked and all their interactions with the computer were recorded.

The learners had a lot of difficulties with the task. They often attempted to carry out tasks, such as logging on to the system, without reading the manual, and complained about the amount of material to read and remember. They lacked basic knowledge, and had problems with terms such as parameter, queue,

pagination. They did not know what was relevant and so were influenced by syntactical connections between what they did or perceived and the problems they were trying to solve. They were also very active in their learning:

"They go beyond the information given by constructing and elaborating ad hoc interpretations of experiences to explain puzzles. They can convince themselves that all is well when what they are noticing is evidence of disaster."

For example, one learner was attempting to enter her password when she made a typing mistake which caused the system to stop and await correction. An indicator light marked "input inhibited" came on. The learner attributed both the delay and light to a heavy work load on the system. Another learner had made an error in issuing a 'file' command and wondered whether her work had been filed. She interpreted the message: INPUT MODE 1 FILE as meaning that it had.

In another paper (Lewis and Mack, 1982) such explanations are described as *abductive reasoning*, where a hypothesis is generated to account for one or more observation. One implication of abduction is that the consequences of errors are explained away and learners do not realise they have made a mistake, and also that they are influenced by superficial resemblences between what they think they need to know and what they see or do. Such abductions are often wrong, and are not tested out. Lewis and Mack (op. cit.) suggest that such reasoning has value in interpreting future events where there is no alternative to abductive reasoning. Often such complex learning is characterised by incomplete and ambiguous information and if learners are to try and understand the process that might lie behind what they experience then they have to use abductions. What Lewis and Mack are in fact talking about is learners building up mental models; which will often be wrong as they are based on fragmented, ambiguous information (as perceived by the learner). Such abductions are triggered by differences between what learners expect and what happens.

Another study in this area is Bott's study of complex learning (Bott, 1979): an attempt to describe and theorise about how people learn about complex bodies of knowledge, in which he includes programming knowledge. He emphasises the need for looking at the process in detail, stating that:

"If one looks at this learning process from sentence to sentence during the instruction, a complicated picture emerges: the process proceeds with backtracks, sidetracks and blind alleys, even in cases where the students appears to be having little difficulty."

A similar point is made by Norman (1983). The basic premise of Bott's theory of complex learning is that all teaching information is interpreted by the student as part of his or her prior knowledge, and learning occurs as a result of the partial failure of this interpretive process, and the subsequent recovery. He agrees with Lewis and Mack, that it is a discrepancy which triggers an inference, or learning: Lewis and Mack focus on the discrepancy between expected and actual events whilst Bott looks at the discrepancy between a student's knowledge structure, and the information he or she perceives.

The domain studied by Bott is that of learning a computer text editor (part of the UNIX system), and all the students were novices. Students were presented with the manual one line at a time on one terminal, and a second terminal was used for the editing task. They were stopped after reading each sentence of the instruction manual and asked to describe what they were thinking about, and often asked additional questions. Like Lewis and Mack's subjects the students had a lot of difficulties. One of the detailed examples given concerns their efforts to understand the print command section of the manual where all of their problems are to do with calling on inappropriate prior knowledge. Students used such knowledge to try and fill in the gaps in their instruction; which Bott refers to as:

"large, "hidden" gaps in the instruction manual with respect to the knowledge actually being taught"

The student's prior knowledge may interact with the knowledge provided and lead to an incorrect knowledge structure being learnt. Bott gives an example of the latter happening early in the instruction process:

"the student's prior knowledge of the word "print" appears to have two separate definitions: (1) to print using a printing press and (2) to print meaning to letter by hand in a non-cursive style. Neither definition is appropriate for how print is used in the instruction manual. Second, the prior concept of "text" seems to centre around a book, or at least a very long manuscript. The prior knowledge of both these words makes it very difficult to interpret "How to print text" And the problems continue with each future occurence of "print", until a suitable definition for this situation is developed...."

In Bott's model, the learner's initial response to instruction is an attempt to automatically assimilate the knowledge in terms of existing knowledge (prior knowledge schemas). When this fails, a second assimilation process attempts to find other prior knowledge schemas which allow a satisfactory interpretation.

Bott also notes that unless there is known information to otherwise explain the adjacency, a pair of events (i.e. two events which follow each other closely in time) are assumed to be causally related, and the noting of a causal relationship between two events sets off a complex assimilation process to satisfactorily explain this causality.

Although this model explains the behaviour of students learning text editing, and would also give a more formal account of the types of behaviour noted by Lewis and Mack, it is not clear how it would account for the development of

programming skill. It is hard to see how all the different kinds of knowledge learnt in programming (including the process of putting a program together) could be learnt by relating it to prior knowledge structures, though these are clearly important.

Summary of section 2.7

Learners experience as many difficulties in learning text editing as they do in programming. The model is of learners who are active in their learning and build hypotheses based on little and fragmented evidence. Bott's model of learning accounts for both his own data and the observations made by Lewis and Mack, but it is not clear how such a model could account for learning to program where there is a large gap between new information received and prior knowledge. The other issue of interest in this section is the methodology adopted. Collecting detailed protocols of learner behaviour is appropriate when the concern is with the learning process (rather than testing performance), and accounting for learner behaviour at a reasonably detailed level. This is also one of the methodologies used in this thesis.

2.8 COMPARATIVE STUDIES OF LANGUAGES

A small number of studies have looked at the issue of transfer between languages: does learning language X facilitate or hinder learning language Y, or have no impact?

A study of Logo and Simper

Weyer and Cannara (1975) studied children aged 10-15 learning Logo and SIMPER, an assembly language interpreter, a simple simulation of an imaginary

machine resembling a Hewlett-Packard model 2000. A curriculum was developed to teach some fundamental programming concepts including concepts such as: machine as a tool manipulated with a command language, machine possessing an alterable memory, name-value associations and evaluation and symbol-substitution. The experimental groups consisted of 8 students learning Logo then Simper, 8 students learning Simper then Logo and 8 students learning Logo and Simper together and an additional two groups learning Logo with graphics. One hour classes were held four days a week.

The researchers planned to use only written materials for teaching, so that the curricula remained totally under their control, but found the task of developing a fully self-contained curriculum for programming very difficult and employed tutors who could help the students over failures in the teaching materials and also report on their interactions. However, the tutors were instructed never to type anything for the students and to encourage them to formulate and try out their ideas before making other suggestions. Developing parallel curricula for Simper and Logo also proved problematic, so the curricula were constructed to teach the concepts in roughly the same order, using whatever features each language possessed that could be best exploited for each concept. Each curriculum was divided into logical parts, which typically discussed more than one concept, and for each part students had programs to work on and fill-in-the-blank questions to answer. Students learning Simper and Logo simultaneously alternately received parts for each language.

The main bulk of data consisted of protocols taken of each student's work, and in addition preliminary aptitude tests were administered, and students filled in questionnaires, and commented on the experiment. Many students familiar with both Logo and Simper perceived Simper as being harder, and this resulted in most

of these students preferring to work with Logo, regardless of the starting language. Unfortunately, few students finished the Logo curriculum, and therefore, group 1 (Logo then Simper) spent negligible time with Simper. However, many group 2 students (Simper then Logo) went far enough with Simper to be able to start Logo, partly motivated by seeing their friends' work. These students who began Logo stayed with it, to the exclusion of further work with Simper. Students given simultaneous access to Simper and Logo chose to spend most of their time with Logo. The authors believe that this was because students could see the advantage of a high level language.

Comparisons of students' error rates are problematic as they had a lot of freedom within the curriculum, which means that the significance of errors varies between students. For example, typing and reading ability varied greatly, so some errors may have been due to differences in these skills; some students adopted a high speed (and high error rate mode of working) whilst others were more cautious, and other students 'stuck' at a particular point in the curriculum, restricted their activities to drawing pictures and so made fewer errors.

Simper

Students did get confused when working with both Logo and Simper, and Logo commands cropped up in Simper protocols and vice-versa. They had problems with the orderly executions of numerals as instructions, and addressing was difficult for many students. The most pervasive problem was mastering the concept of context (or locality of information) both from the student's point of view as user and from the point of view of instructions within his or her programs. Students did not fully grasp the distinction between editing commands and machine/assembler instructions. Toward the end of the curriculum, procedures and their calling sequences provided examples of how programs could be structured by writing functionally related subunits, yet students failed to structure

these programs correctly. None of the students had time to do any significant work on the final part of the curriculum dealing with stacks and recursive procedures and so there is no data on this. Much of the data discussed was from the more able, and typically older, students.

Logo

Students sometimes forgot to quote literals; they reversed name and value positions or they attempted linked assignment through one command. The fact that Logo allows numbers to be names also led to some confusions between literals and names, and actual and formal parameters. Many of the examples given of students' programs do not give the impression of students with a very deep understanding of Logo. In one group of problems, students forgot to put input names in the procedure title, used literals in place of names, or used names different from those named in the title. They often failed to break down problems into manageable parts, and therefore did not notice that some of the components had been solved previously.

Although the results are interesting, they cannot be said to be robust, or generaliseable for a variety of reasons. This experiment was in part a pilot study, and both the Logo and the Simper curricula were changed 'on the fly' during the course of the experiment, if student reactions and problems suggested that a change was needed. The students did not finish the Logo curriculum, and spent relatively little time on Simper. Within the constraints of the curricula it appears that they were free to spend time on what they were interested in: this also meant that Logo was pursued at the expense of Simper. The three groups were all relatively small in size (8), and there was some drop out. While the authors stress that qualitative methods are appropriate, they do discuss errors per command for each student, but their section on understanding the students, which is the main thrust of the report,

gives no indication of how widespread or pervasive the problems discussed above are. The interest in this study is that it is one of the few studies which have looked at the sequential learning of languages, and given the qualifications mentioned, students were able to learn Simper and Logo together: the authors state that: *"students can learn both languages, nearly simultaneously, and do so faster than students who learn the same languages sequentially".* However, it is not clear how deep such understanding goes, so this has to be seen as a claim backed by weak evidence.

Learning Pascal after Basic

A study by D'Arcy (1985) has investigated the problems of learning Pascal after BASIC. Here the issue is not whether it is possible to learn two languages together, but whether learning one particular language which has been heavily criticised, particularly for its lack of structure, has a detrimental affect on learning a subsequent language. D'Arcy reports several informal observations that this is the case. His own research compared a group of Pascal students with previous training in BASIC with a group who had no previous experience on a number of comprehension and modification tasks, and his data shows little difference between the two groups overall. Possible reasons for the lack of negative transfer may be that the tasks administered were too easy (with around 80% of both groups being successful); and that comprehension and modification tasks may not be adequate measures. Another possible reason is that the course was well structured with emphasis on good programming technique rather than syntax.

Learning Prolog after Pascal

White (1988) is studying mapping failures in analogical transfer. Analogical learning is discussed in the next, more theoretical chapter, chapter 3, and as White points out, there are problems in determining the appropriate correspondances

between the base domain (what is already known), and the target domain (what is being learnt). As part of a preliminary study White has investigated the effect of having learnt Pascal on learning Prolog.

He studied students with a good knowledge of Pascal, and who were studying a Prolog course. The students' task was to predict the behaviour of simple programs, concentrating on the value of the variables. They worked on a number of programs: first of all predicting the behaviour of Pascal programs and then predicting the behaviour of the Prolog programs. They were required to explain the programs' behaviour to fellow students and also tested their predictions by running the programs.

In working on the Prolog programs, all but one of the students showed evidence of Pascal - type misconceptions, and the effect was very strong. However the protocols suggest that the students did have a suitable model but did not select it initially. As yet it is not possible to draw firm conclusions from this work which is still in progress. Further work is needed to investigate how the programming knowledge is structured and which parts of the structure are transferred.

Summary of section 2.8

Altogether then, from these studies there is little empirical evidence of the positive transfer of programming skills between different programming languages. The preliminary investigations by White suggest that students with knowledge of Pascal may use this to build mental models which are inappropriate for understanding Prolog problems. The Weyer and Cannara study demonstrates the possibility of learning two languages almost simultaneously, whilst D'Arcy's research suggests that BASIC need not "mutilate" given careful course planning: however, even this is a cautious suggestion given the limitations already mentioned of the methodology. Given that a substantial proportion of novices will go on to other

languages after their first, this remains an important area for investigation.

2.9 SPECIFIC LANGUAGES: SOLO AND LOGO

This section discusses the available literature on learning the specific languages that were used in this study.

SOLO

SOLO is not a widely available language, as it was developed for Open University students taking a cognitive psychology course (Eisenstadt, 1978). However, there have been a number of studies of students using SOLO in this context, and on tutoring aids for use with SOLO.

Lewis (1980) analysed a large number of programs written by SOLO students, and Eisenstadt and Lewis (1985) report on their experiences over a four year period of designing, implementing, testing and iterative re-design of SOLO and accompanying materials. As part of this the programming activities of 96 students were analysed, and their errors categorised. The greatest proportion of errors were syntactic: e.g. 34.4% Spelling correction and quote balancing; 25.3% wrong number of arguments passed, 9.5% invocation of non-existent procedures - although the latter two errors could have arisen from a number of causes. In fact Eisenstadt and Lewis call these categories symptoms and attempt to identify the cause of these symptoms from the context surrounding the errors. For the first three given above there are 19 causes. This analysis led to a number of changes to SOLO which in turn have led to SOLO users making more interesting errors.

Kahney (1982) studied a large number of SOLO programmers engaged in a variety of tasks. 130 programs designed as solutions to a particular programming problem were analysed, and despite great surface variety, there was considerable underlying order in the programs, most of which can be classified as one of five types. It is suggested that novices experience a lot of difficulty making their programs reflect the mental models they have in their heads, and moulding data structures to program designs is one solution.

In a different part of the study, concept sorting and recall tasks were used to induce novices' mental organizations of programming knowledge. The important finding here was that there was no significant differences between novices and an 'ideal' expert with respect to the underlying organizations of conceptual (domain related) knowledge. Novices' mental models of the behaviour of recursive programs were determined from their selection of programs in a questionnaire where they were asked which of three programs would produce a particular required outcome. Very few students had an accurate model of recursion, and others had a range of models ranging from what Kahney calls "null" models (i.e. none at all) to loop models. A further experiment looked at the structure perceived by novices when they are presented with unfamiliar programs. In this task subjects were presented with unfamiliar programs for which a transcription task was devised. On each trial subjects were required to use a different coloured pen so that the amount of information obtained and remembered on each viewing could be ascertained. Generally the novices extracted information from these programs a line at a time whereas the expert used the structure of the program.

Another problem given to subjects had elements designed to activate the programmers' real world knowledge and other elements designed to activate programming knowledge. The data from this task was used to develop a model of

problem solving for ill defined problems. An abstract version of this problem was also designed, which none of the 9 subjects were able to solve, nor were an additional set of subjects able to solve the problem when a 'clearer' version of the abstract programming statement was given. From these two experiments, an interactionist theory of problem solving by programmers was devised which was tested on the same problem, and a production system model of the coding and program evaluation phases of one of the subjects was also developed.

The focus of this work is on the knowledge novices possess and how this is used in solving programming problems. The protocols provide some support for the interactionist theory which is posited, but the theory has only been tested in a very limited situation where integrated, domain related knowledge already exists in the minds of programmers. Kahney describes novices as problem solvers who have acquired "'more or less' intelligent domain related schemas". There are three interesting outcomes from this work. It shows that the differences between novices and experts cannot be summarised as experts having more, better organised knowledge: there can be overlapping behaviour and some novices can behave more like experts, so the path of acquiring expertise needs to be studied. Secondly, the detailed work on understanding recursion points to the difficulties that novices have, and the impoverished models they develop, if, indeed they develop any. Finally it indicates the viability of modelling such behaviour, though it does not go very far in this direction.

Hasemer (1983) developed an intelligent debugging system for SOLO students. The basis for the system is an explicit account of how experts debug faulty code, and this account does not rely on the intentions of the programmer, so the system can work from the code produced. In the process of doing this, a broad taxonomy of naive users' errors is given, and it is this which is of most interest here. The four categories of errors used are: simple syntactic errors, higher-level syntactic

errors, cliche errors and data flow errors. Simple syntactic errors arise through misusing the language's syntax at an elementary level - at a level of "words". Higher level syntactic errors are mistakes at the level of programming constructs. They could be mere slips of the expert's attention but for the novice may well indicate a more serious lack of understanding. Because the significance of the error will depend on the expertise of the user, Hasemer has avoided the usual distinction of semantic and syntactic errors, and higher-level syntactic errors could well be symptomatic of semantic problems.

The SOLO system used by Hasemer had changed quite considerably from the system used by the students which Lewis reported on so it is difficult to make comparisons. Hasemer points out however that although the overall proportion of errors remains similar, their distribution is rather different. The example he gives is of the "unbound variable" error moving from the bottom of Lewis's list to near the top of MacSOLO's - the system used by Hasemer. This suggests progress in that students are making more interesting errors, - the student of MacSOLO is attempting to use general programming technique whereas the errors analysed by Lewis suggested an attempt to get to grips with the basic SOLO machine. The error lists suggest that the aim of obviating "silly" errors appears to have been achieved.

The cliché errors identified by Hasemer are those in connection with what he calls "conceptual 'chunks' of code" - elsewhere referred to as plans (see section 2.4). The final category is data-flow errors. An example of this would be when the code is syntactically correct but is given the wrong data to work on. The model of debugging presented consists of three stages: 1) skimming the faulty code looking for salient points, i.e. syntactic errors; 2) rechecking the code looking for errors in higher level segments of it, here identified as programming cliches, and 3)

rechecking the code again, this time following data-flows, and identifying the code in terms of the program's overall purpose, if known. The programming cliches are interesting from the point of view of novice behaviour because the errors novices make can be analysed in terms of their plan behaviour. For SOLO, Hasemer has identified 9 cliches which are used by his debugging system AURAC.

Logo

Some of the studies of Logo have already been reported on in section 2.7. This section will briefly review some of the work not already covered. The previous section discussed work by Eisenstadt and Lewis on SOLO: as part of this they made a direct comparison between the four most common errors in SOLO and Logo, making allowances for differences between the two systems. This is reproduced below:

	SYMPTOM	% OF ALL ERRORS	
		Logo	SOLO
1	Spelling/typing/misquoting	28	34
2	Wrong number of arguments passed	18	18
3	No line number	12	9
4	Call to undefined procedure	12	9

Most recent studies of Logo are classroom based, and often focus on different aspects of learning Logo. Most of the work in the U.S.A, has been investigating whether learning Logo has general cognitive benefits, and the results of these studies were reported in section 2.6. A related piece of research but tackling a rather different theme is Heller's work on different Logo teaching styles (Heller 1986). She investigated the differences between a group of children receiving structured Logo teaching and experiential teaching, and found that the group learning in a structured environment scored higher on a test of Logo content.

Carver and Klahr (1986), also in the U.S.A. have studied children's Logo debugging skills. This study is particularly interesting because the authors have developed a formal model of debugging skills which is then used to assess which debugging skills are typically acquired by children learning Logo. The particular domain is turtle graphics, and the model has access to the goal drawing (i.e. the intended outcome), and the drawing that the buggy program produces. Discrepancy is defined as the difference between the goal drawing and the program's output, and the bug is the erroneous component of the program which caused the discrepancy.

The model has four stages: 1) evaluation where the program is run, and the output is compared to the goal drawing, so as to determine whether debugging is necessary; 2) identification of the bug by using descriptions of discrepancies between the goal drawing and the program output to propose specific types of bugs which might be responsible; 3) bug location where clues about the structure of the program are used to narrow the search, and 4) correcting the bug, and retesting the program. The model is implemented in GRAPES, a goal restricted production system, and can either work with high-information, where the user supplies a lot of knowledge about the probable type and location of bug, or with low information, where the user does not supply this information. Not surprisingly, in the second case, the program is much less efficient at diagnosing the bug, as the search space to be narrowed is much larger.

The model has been used as a formal representation of the debugging process, against which children's skills could be assessed. Nine children attended 12 two hour Logo classes over a three week period. They were tested at various points during this time. Although they learned the editing and command generation skills which are a prerequisite to debugging, they were not able to interpret commands

and use clues to identify, locate and correct bugs. They could, however, correct bugs once they were identified and located. They rarely debugged programs when not required to do so, preferring to start again. When debugging was required they searched through the program listing for the bug. The authors conclude that the children fail to acquire debugging skill because it is complex, because it requires extra capacity (which they haven't got), and because it is not explicitly taught.

Carver then developed a curriculum to teach the specific components necessary for debugging directly (Carver, 1986) and used it in a Logo list processing course (Carver and Risinger, 1987). The students' debugging speed and efficiency improved as a result of learning to narrow their search for bugs: they operated more like the high-information version of the model discussed earlier. A further interesting finding was that the children who acquired effective debugging skills in the Logo course improved more on transfer tasks involving debugging writing instructions than students who had not taken the course. This is one of the few positive transfer results in this area and the authors suggest that this is due to the careful task analysis - no doubt helped by the explicitness of developing a formal model, and explicit debugging instruction.

In this country the focus on Logo has often been on its use in the context of learning mathematics. The early work at Edinburgh was in this vein, (Howe, O'Shea and Plane, 1978) as is the work carried out by Hoyles and colleagues, e.g. Hoyles and Noss (1985). Where programming skills have been assessed, the findings are similar to those from the work in the U.S.A. For example, one of Noss's studies was a component of the Chiltern Logo project in which the Logo work was integrated into the curricular activities of the classroom, and the children worked in small groups or in pairs (Noss, 1987). The childrens' programming activities consisted of two phases, the second of which was the

programming phase which lasted seven months. During this time a number of key concepts were identified as important landmarks in the children's acquisition of skill, and it was also noticed that particular kinds of mathematical behaviour were associated with specific types of programming activity. These 'learning modes' were proposed as providing some insight into the nature of the childrens' mathematical and programming activity, and both the key programming concepts and the learning modes were used to structure the analysis. The programming concepts employed as a basis for analysis were: procedure, iteration, subprocedures, editing and debugging, inputs and recursion. The children's ability to use these concepts varied enormously: for example of the five schools studied, recursion was only used successfully by all children in one group: in the other groups it was used by an average of less than 50% but this ranged from 33 to 67%, whilst understanding of edit/debugging varied from 23 to 90%. However, there was a wide disparity between classes in the amount of programming time available. The rest of the analysis focuses on the style of learning and learning strategy, and three "learning modes" are hypothesised which are: making sense of a new idea, exploring, and solving problems. Although Noss was focussing on mathematical learning rather than problem solving skills, the failure of many children to acquire more than rudimentary skills supports the findings of the studies reviewed in section 2.7.

Summary of section 2.9

The work reported above gives a very good empirical base for looking at SOLO novices. The data is, however, limited in a number of ways. The work by Eisenstadt and Lewis was geared towards the design of the system, and eliminating as many errors as possible: *"An analysis of the behaviour of the group of students leads us to develop the notion of pre-emptive design, in which we try to trap or pre-empt virtually all of the confusing problems which plague novice*

programmers".

The success of this approach is shown by the fact that during the course of SOLO's development students became more sophisticated and made more interesting, semantic errors than the trivial syntactic errors made previously.

The work of Kahney has looked specifically at problem solving behaviour and Hasemer's concern was to build an intelligent debugger. None of these studies, therefore has studied in detail the problems novices have as they develop their skills, working through the SOLO curriculum. Some of the work on Logo was reported in section 2.6. The research discussed here supports the conclusion offered earlier that only modest competence is achieved.

2.10 SUMMARY AND CONCLUSIONS

The aim of this thesis is to research the problems that novices have when they are working in environments which provide good conceptual models. This section summarises the findings which have been discussed in this chapter, assesses how far the findings help to achieve this aim and points out some of the gaps.

Section 2.2: programming and natural language constructs

It is clear that how languages are designed and structured can make a lot of difference to the ease of learning to use them and in understanding them. The work of Green and colleagues shows that different types of languages facilitate the extraction of different types of information; that the structure of languages can help in using conditionals and that comprehensibility depends on getting a match between the structure of the language, and the task being performed. Such

findings can inform design, and therefore affect the ease of use, and indeed design guidelines are offered. As far as novices are concerned, however, some problems will still remain as much of the work reviewed on notational features is geared towards professional programmers, whilst this thesis concentrates on the problems that novices have. Turning to 'natural language', the mismatch between natural language and programming languages may explain in part why programming is so hard, but does not result in a "best bet". There is no solution such as basing programming languages on natural languages, and languages which do not have a procedural emphasis, such as Prolog, attract their own special kinds of problems.

Sections 2.3 and 2.4: experts and novices, and plans

There is general agreement that expert programming skill can be viewed as a large amount of organised programming knowledge and rules for putting it together, and knowing when to use which bits. The evidence on how novices' and experts' knowledge differs is less clear, but there is some consensus that novices and experts categorise what programming knowledge they do have differently. Plans are clearly one important way of organising such knowledge for experts, and have been used as the basis of bug classifications and are used in comprehension. There is some evidence of their use by novices and some of the errors made by the novices studied in this thesis are analysed in terms of plans. Nearly all of the work on plans, however, has been on Pascal, and it is not clear how generaliseable they are. Whilst Gilmore's work (which is the most comprehensive study of plans) found no evidence of their use by BASIC programmers, Davies's replication study indicates that plans can be used by BASIC programmers, if they have been taught in a structured way. There is also some support, albeit weaker, for their use in other languages, for example LISP. As far as novices are concerned this work focuses on the skills that novices haven't got. The main contribution of this work, therefore, is to increase our understanding of the nature of programming skill, to emphasises the importance of plans for experts - and possibly novices, and to

outline the limitations of plans as forming the fundamental structure of programs. All this is necessary but not sufficient. To understand the learning processes of novices, they need to be studied on their own terms.

Section 2.5: conceptual and mental models

Conceptual models are important because it has been suggested that using such models in instruction will give novices a framework for relating new information to their existing knowledge. This work is particularly important for this thesis because the novices who are being studied are learning programming with the help of conceptual models. Du Boulay, O'Shea and Monk provide some prescriptions for designing instruction around a conceptual model, but this is not empirically based, and there has been no empirical investigation, hitherto, of the prescriptions they suggest. Mayer's work has demonstrated the efficacy of conceptual models in general, although they are most helpful for less able learners or cases where the instructional material is poorly organised. The work in this area is also of interest because of its focus on instruction, though it does not analyse existing curricula.

Section 2.6: children, problem solving and field studies of novices

The field studies of children learning Logo and BASIC, reported in section 2.6 are of particular interest as they concern how children learn on typical programming courses which are closer to the conditions experienced by the learners who were subjects for this thesis than the experimental situations looked at in previous sections. However, the curricula were not designed around conceptual models. The main result throughout these studies is the lack of competence acquired by most of the students on typical programming courses. Another finding is the lack of support for the idea that programming fosters the development of powerful problem solving skills. Pea and Kurland explain this partially by the low level of exposure to programming on most programming courses. They suggest that one

route forward is to focus on the transfer of "low level" skills. In part of this thesis this route has been followed, and empirical studies were conducted which looked at the transfer of low level skills between two programming languages.

Section 2.7: learning to use complex devices

The studies reviewed here indicate that learners have as many problems when learning text editing as they do in programming. They are active in their learning and build hypotheses based on little and fragmented evidence. Bott's model of learning accounts for both the data he has collected and the observations made by Lewis and Mack, but it is not clear how such a model could account for learning to program where there is a large gap between new information received and prior knowledge. The other issue of interest in this section is the methodology adopted. Collecting detailed protocols of learner behaviour is appropriate when the concern is with the learning process (rather than testing performance), and accounting for learner behaviour at a reasonably detailed level. This is also one of the methodologies used in this thesis.

Section 2.8: comparative studies of languages

Two of the three studies reported here which have been concerned with second language learning report positive results. Weyer and Cannara's study indicates that children **can** learn two languges simultaneously, but this is not a strong finding, as there are various problems wiith this study. D'Arcy's study of students learning Pascal after BASIC shows that students need not be 'mutilated' by exposure to BASIC as their first langauge. The third study, by White, is still in progress, but there is some evidence that previous knowledge of Pascal may lead students to make inappropriate analogical mappings when they are trying to understand Prolog programs.

Conclusions

There are a few areas which have not been developed so far, and which are of interest to the aims of this thesis. As has been mentioned, du Boulay, O'Shea and Monk's prescriptions are not empirically based, and although it is not the main aim of this thesis, the work done here does evaluate these guidelines. More importantly, very little work has been carried out on analysing and evaluating the instruction used, yet it is clear (and unsurprising) that this plays an important part. One of the few studies to achieve any positive transfer effects is Carver's study which analysed and explicitly taught debugging skills. This thesis will analyse the instructional design of curricula used in the context of providing conceptual models. It takes up Pea and Kurland's suggestion of focussing on "low level" skills, and like the work reviewed in section 2.7, focusses on the process of acquiring programming skills. The decision to look at students learning low and high level languages sequentially in this thesis brings two strands of work together. It extends the work of Weyer and Cannara, which is interesting, but provides rather weak evidence, and it does so in the context of learning with the help of well formulated conceptual frameworks. Finally, the students studied are working with self-study texts, and so the results of the studies will be of particular interest to designers of distance learning materials in this domain.

Chapter 3

THEORETICAL BACKGROUND

Contents **Page**

3.1 INTRODUCTION

This chapter provides a conceptual framework for the thesis. The first section begins by considering the three theoretical influences on the study, which are Piagetian theory, artificial intelligence and educational technology. The role of analogies in conceptual models in a number of domains is considered, and various examples given, and the application of such work to teaching computing concepts is discussed. It is suggested that the criteria offered by du Boulay et al (1981) as guidelines for using conceptual models to teach novices are not sufficient, as they concern the nature of the conceptual model, whereas it is also necessary to pay attention to the presentation of the conceptual model. In this respect, it is argued that there are three further ways of describing conceptual models: in terms of state, function and procedure, each of which may be emphasised to a different extent. The three aspects are defined and discussed.

3.2 THEORETICAL BACKGROUND

There are three distinct influences on this study which all converge: Piagetian theory, artificial intelligence, and educational technology. One starting point is broadly Piagetian, in that it is assumed that learning involves assimilating new information into existing structures, and that new information connects with existing organised knowledge. Piaget emphasised the need for knowledge to be structured. He claimed that *"to know an object is to act on it"*, (Piaget, 1964) and introduced the idea of an operation, which is an internalised action that is linked to other operations and part of a total structure. *"Such operational structures.....constitute the basis of knowledge."* (Piaget, op. cit.) His stage theory, is in fact the development, or transition from one set of structures to

another. Such knowledge structures enable us to assimilate new information, which is incorporated into the existing structures, which in turn alter to accommodate the new knowledge. However, if there is a large discrepancy between the new knowledge and the structure that exists, such that the structure cannot accommodate the new information, there is a development to a new structure: to the next stage of development.

The basic principles of this theory still underlie much of the research carried out on both children and adult learning. However, there is a particular problem for adult learning, in that Piaget emphasised development as the primary spontaneous process: "the development of knowledge is a spontaneous process tied to the whole process of embryogenesis" (which concerns the development of the whole organism). However, embryogenesis ends in adulthood, and so cannot be an acceptable basis for learning in adults. Piaget emphasised the subservience of learning to development:

"learning is subordinated to development, and not vice versa" (Piaget, op. cit.).

It is this point which is the main departure point between Piagetian theories and many Artificial Intelligence theories of learning which do not uphold this distinction between development and learning.

One of the arguments in favour of introducing novices to computing via conceptual models is fundamentally based on Piagetian theory, as the argument is that as computers are so different from other devices (a point taken up in section 3.3), there won't be an existing framework into which such knowledge can be assimilated. For example Mayer (1980) says:

"Novices tend to lack domain-specific knowledge.....Thus one technique for improving the novices' understanding of new technical information is to provide them with a domain-specific framework that can be used for assimilating new

information....."

The second influence on this research is artificial intelligence. Artificial intelligence (AI) has contributed to the field of problem solving in a number of ways. The first of these is the development of methodologies for studying problem solving (such as protocol analysis) which can be the first stage of constructing models of human cognition, (for example, Newell, 1977). This is discussed further in chapter 5. The second AI contribution is that such computer models provide a way of testing our theories of human cognition: they are an aid to theory development. Moreover, developing a theory which can be "tried out", i.e. forms the basis of a computer program, requires the theory to be explicit. This has led to a focus on learning and problem solving at a detailed level, e.g. Brooks' work mentioned in chapter 2 (Brooks 1977), and this level of detail is helpful when we turn to consider the issue of instructional design. There has also, however, been a great interest in learning, and specifically in the representation and structure of knowledge: the main reason being that when modelling human cognition it is necessary to start with a lot of knowledge and represent it in an appropriate way such that it can be used and modified.

Artificial Intelligence theories of learning often have an uncanny similarity to the ideas of Piaget. An example of this is the role of schema theory in memory, where schemata can be thought of as packets of information stored in memory and representing general knowledge about objects, situations, events or actions. An important characteristic of schema theories is that each schema is linked into related systems, and an overall schema may consist of a set of subschemas. (Rumelhart and Norman, 1983). As with Piagetian theory, the fact is emphasised that what we remember is influenced by what we already know. A specific example of such a theory is Schank's model of dynamic memory (Schank, 1982). This builds on the work of Schank and Abelson on scripts, (1977) but in order to

account for reconstructive memory, and the ability to generalize and learn from past experience, Schank proposes the existence of memory structures which he calls MOPs, - memory organisation packets. The notion of scripts is not abandoned altogether, but it is redefined within the new model. The point is that this notion shares with Piaget an emphasis on the structuring of knowledge in structures which are modified in the light of new experience.

The third contribution of AI arises from the second: in the same way that constructing programs that model human cognitive processes is viewed as a way of understanding those processes, it is argued that through programming students learn about the problem solving processes because the very process of programming forces the stages of their problem solving to be made explicit. AI researchers have therefore developed programming languages such as Logo, and claims have been made that the benefits of learning Logo extend beyond acquiring the skills of programming (Papert, 1980)[1]

The third major influence on this thesis is the practical problem of instructional design. One of the major issues in the Institute of Educational Technology at the Open University is how to improve instructional design, (see Jones, Scanlon and O'Shea, 1987). One approach is to collect protocols of student behaviour with the intention of modelling such behaviour, and to use the models either in designing computer tutoring systems, or in improving the curriculum. An example of such work is Scanlon's work on novice physics problem solving (Scanlon, Hawkridge, and O'Shea, 1983.) Unlike Scanlon's work, the work in this thesis is not aimed at producing computer models, but does have much in common with this approach in terms of collecting and analysing student protocols. Given the practical nature of programming, teaching programming at a distance may have special problems,

[1]And see the discussion on cognitive effects of learning to program in chapter 2

and there has been an interest in how best to teach computing concepts to novices. An example of the application of work in this area has been the Micros in Schools project (O'Shea, 1984; Jones, 1986; Jones and Preece, 1984) which adopted certain design criteria which aimed to both guide students through material at a keystroke level, and explain the rationale for each step. One motivation, then, for studying learners' cognitive processes at a detailed level is to gain an understanding of problems from the learner's point of view, that can inform curriculum design.

3.3 WHY COMPUTERS ARE SPECIAL

Computers are fundamentally different from other machines. Consider a simple photocopier[2], and "understanding" it. There is the case of not understanding the idea of copying a document, the very concept of copying. There is also the possibility of not understanding how it works sufficiently to debug a problem, or use it efficiently. An example of this is using it with headed paper. Most people don't know, or remember which side of the paper the photocopier will print on and which is the "top" and the "bottom" of the paper and therefore have to experiment. However, there is no question about the types of things it will do, i.e. its function, although one might be unsure about how sophisticated it is, e.g. can it sort multiple copies? Machines such as vacuum cleaners or washing machines are similar in that the problems they can solve (their function) is relatively clear, and also what they can't do (for example wash the car). Because a computer is a general purpose device, however, for many people, and especially novices, the class of problems that it can deal with is unknown. There may be a number of

[2]The argument that follows is limited to simple photocopiers as opposed to the more recent programmable devices which in terms of this debate are similar to computers, and similar problems are encountered in their use.

options for operating a copier, - for a computer, there is an undefined set. The biggest difference however, is that in a copier you don't need to specify the instructions in a meta language, whereas in a computer you have to do just this.

It is hard, therefore, for the novice to make comparisons with devices that she does know about: there is no 'naturally existing' framework that will serve to relate new information to existing knowledge. It will not be possible to use analogies with machines that the novice does know about, and be sure that these analogies are appropriate. The next section looks at analogies and metaphors, with a view to their use in conceptual models, and their role in the learning process.

3.4 ANALOGIES AND METAPHORS

The terms analogy and metaphor tend to be used interchangeably. The literary definition given to most of us in school is that analogy compares two things by stating that they are alike: *the sun is like a ball*, whereas a metaphor asserts an identity relation rather than a comparative one: *the sun is a ball*. This definition isn't very helpful in using analogies and metaphors for learning, as it doesn't take the learner into account. Gentner (1983) offers another definition: Analogy makes the basic mapping explicit, whereas in metaphor the mapping must be recognised or reconstructed by the understander, thus *Juliet is the sun* and *Juliet is like the sun* would both be metaphors (unlike the previous definition), as it is not made explicit in what **way** Juliet is like the sun (round? hot? made of fire?) This is more helpful for our purposes: we can see that saying a variable is a box, or a computer is a filing cabinet leaves open the question of what the relationship is. However, it should be noted that within cognitive science, the term analogy is often used to describe situations where the relationships are clearly not explicit, and so in this thesis it will be assumed that either term can apply to a situation

where the relationships are **not** explicit, with the exception of Gentner's work in section 3.6. Before coming back to the use of analogies and metaphors in teaching computing, a brief account is given of the literature in other domains, where there there has been more work carried out on their use.

3.5 OPERATIONAL THEORIES

Operational theories of metaphor focus on how metaphors operate: how they are used, and the effects of suggesting metaphoric comparisons to learners. For example Carroll and Thomas (1982) claim that the activity of learning to use a computer system is structured by metaphoric comparisons: they suggest that learnability depends on the metaphors suggested and how helpful they are. They suggest an account of consolidation and interpretation where the material to be learned is entered into working memory by hypothesis, a framework of related general knowledge is retrieved from long term memory and also entered into working memory. When further new material is encountered there is the need to consolidate and compress the contents of working memory into a more integrated format, and this is done by assimilating the new material into the retrieved frameworks. This account is very similar to Bott's model which was discussed in chapter 2.

Lakoff and Johnson (1980) have written about the use of metaphor in everyday life. Their argument is that metaphor systems are so deeply embedded in everyday language and culture that we tend to be unaware that they are not literal. Few of us, for example are aware of the metaphor when we talk of "*higher prices*". It is argued that metaphors do not occur singly, but cluster within systems, and such systems can be self contained (and internally consistent) but different systems may

differ. Thus one system is the "good is up" metaphor, containing phrases such as "I'm feeling high (or low)",..."elated", "spirits are soaring", "things are looking up", "the market's depressed". A different system embodies the "more is up" metaphor as in "prices are soaring". Note that systems may sometimes conflict: here "more is up" is in conflict with "good is up" (higher prices are usually not good in this context.)

Operational accounts such as that of Carroll and Thomas, mentioned above, do not specify the mechanism by which the process happens. Which parts of the new information are mapped onto the old, and how? Learners generate metaphors spontaneously: how do they decide whether two systems are analogous?

3.6 STRUCTURAL THEORIES

Gentner (op. cit.) has investigated the use of analogical models in science. She analyses metaphor in terms of primitive pattern matching operations between the elements and relations of structural descriptions. She has developed a theory of structure mapping from a (known) *base* domain to an unknown *target* domain, which is represented as semantic nets. For example she analyses the Rutherford solar system model of the hydrogen atom, in which the object-nodes of the base domain (the sun and planets) are mapped onto object-nodes of the atom (the nucleus and electrons). Given this correspondence, the analogy conveys that the relationships that hold between the nodes in the solar system also hold between the nodes of the atom; for example, that there is a force attracting the peripheral objects to the central objects; that the peripheral objects revolve around the central object; that the central object is more massive than the peripheral objects; and so on. Using this structural description and mapping she is able to classify various

properties of the mapping, which can then be used to assess the likely usefulness of the metaphor. For example:

Clarity: A mapping has clarity when a node from the base domain maps to one and only one node in the target node and vice versa. In the case of the Rutherford example, the objects-nodes are the sun and planets in the solar system, and the nucleus and electrons of the atom, and the sun maps onto the nucleus (and only the nucleus) and the planets map onto the electrons, and only the electrons, and so the mapping has clarity.

Richness: This is the amount imported from base to target, i.e how many nodes map? In the Rutherford example there are two.

Systematicity: This is the degree to which relations mapped also belong to a known mutually constraining conceptual system. In the Rutherford model, which is highly systematic, the mapped relationships- ATTRACTS (sun, planet), ORBITS AROUND (planet, sun) etc. form a connected system.

Abstractness: This refers to the kinds of things that are mapped: relations or attributes.

She suggests that to be useful in science, analogies should be high in clarity, for explanatory power and systematicity, and low in richness, but in literature they should be high in richness and low in clarity. These suggestions are based on predictive power which is important in science but not in literature. It is also important for conceptual models of computers.

3.7 OTHER APPROACHES TO LEARNING BY ANALOGY

In computing domains work has been carried out by Carroll and Mack (1982) on word processing and Bott (1979) on text editing. There are a variety of problems with such a structural analysis. For example, it is not clear how it is decided what primitives should be included: in the Rutherford metaphor another salient relation which does not carry over is "sun warms the planets". By not including this relationship the mapping appears better than it is, and also for the purpose of using the analogies to make predictions, the learner needs to know which relations hold and which don't. Carroll and Mack (op cit.) make the point that the system falsely classifies analogies that patently work. For example, according to Gentner's definition the analogy of water waves to sound waves is not good, because it is not transparent, (a property which was not included in the above list), yet it is manifestly good in that it is established and works well. They also point out that structural theories focus only on similarities, yet it is sometimes the disparity between the metaphor base and target which can be salient and can lead to insights. Like the operational approach, it does not specify the processes by which the corresponding nodes and relationships are recognised.

Carroll and Mack advocate a complementary approach which aims to move towards specifying such processes. They emphasise that metaphors are inherently open-ended, by which they mean that the learner plays a role in determining whether some of the relationships hold. Metaphors can serve to create a contradiction in a set of beliefs whose resolution leads to new understanding. They give the example of the 'text-editor is a typewriter' metaphor inviting :

"active and creative thought on the part of the learner that may go well beyond simply noting the matches and mismatches between the elements of one object and

another."

In other words, metaphors can orientate the learner to hypothesise and verify similarities in structure and function and give a framework for recognising and analysing discrepancies. According to this view, the relevance of a set of possible mappings depends on the needs and goals of the learners, - so for example, the possible "warms" relation in the Rutherford metaphor (sun warms planet) is not relevant because it is not relevant to the goal of the learner which is to understand the spatial properties of the hydrogen atom. They also point out that learners are active, they engage in generating and testing hypotheses and will generate metaphors. Carroll and Mack, therefore, see their approach as complementary to the operational approach which is practical but insufficiently principled, and the structural approach which is too theoretical.

Two points are important for computer domains. One arises from the work of Bott who argues that a student faced with incomplete knowledge and analogical structure will attempt to assimilate all new information into the analogy. His example of the nearly right match was given in chapter 2 which suggests that when learners are generating their own analogies, and the teacher has therefore not got access to them, analogies which match very closely may be less helpful than those which match a little less closely. This is because the user's experience is that the two systems are the same, and can develop pervasive mental models on this basis, which are hard to debug, as they fit 99% of the time, yet the other 1% may well be crucial. According to Bott, such problems will not arise with given metaphors because the teacher can point out exactly where the analogy breaks down, to avoid the pervasive but wrong models. Even here, however, if Carroll and Mack are correct about metaphors being necessarily open, misunderstandings can occur. This leads to the second point which is that as learners are active, and will make

wrong inferences, they need to be provided with tools which help them to debug their erroneous models, and thus recover from their errors.

Another different approach to learning by analogy is Holyoak and Thagard's computational model of analogical problem solving (Holyoak and Thagard, 1989). In this system, problems can be solved either by the direct use of rules using bidirectional search, or else by the activation and use of analogies with prior problem solutions. The model attempts to embody the opportunistic use of analogy within a rule-based system. Their general aim is to:

"*integrate a set of learning mechanisms with a set of performance mechanisms for solving problems and generating explanations.*"

Analogical problem solving is viewed as being composed of three stages: first of all accessing a plausible analog in memory, then adapting the source analog to the requirements of the target problem, and finally inducing a new knowledge structure that summarises the useful commonalities between the source and target. The first step is of particular relevance to the use of analogy in learning, as one important issue is how does the learner (or teacher) decide what is a useful analogy, and how the mapping takes place.

This model assumes a cognitive architecture which needs ways of representing simple factual (and propositional) information such as "Lassie is a dog", and also ways of representing general rule-like information such as "dogs have fur". It is also necessary to represent general concepts such as "dog" and "fur" and to provide structures for representing complex problems to be related by analogy. The model, "PI", uses structures called concepts to integrate these kinds of knowledge. Unlike Gentner's structural representation, it has some knowledge about the source domain. Two basic types of search process are used: forward search from the problem situation to possible actions, and backward search from the goal to possible actions and pre-conditions that could achieve it, and a spreading activation

process to retrieve information which is related to the current situation. A spreading activation search is also assumed by Bott and by Carroll and Mack (op. cit.)

At present PI has only been used to model analogical problem solving in one domain, the "radiation problem".[3] Subjects find this problem very hard but their performance is much better when they are told of an analogous problem, the fortress problem.[4] One solution is to split up the army and attack the fortress from different directions. Analogously, the tumour can be eradicated with lower intensity rays from different directions. Gick and Holyoak (1980) found that 75% of their subjects formulated this solution after hearing the story analogy and after being given a hint, but only 10% generated a solution when these conditions weren't met.

PI is able to run a highly simplified simulation of the solution of the radiation problem using the fortress analogy. Determining which relations can be mapped requires the knowledge that the system has about the nodes and relations. For example, an association must be made between DESTROY in the tumour problem and CAPTURE in the fortress problem:

"This association is found as the result of subgoaling, using rules that produce the following chain of subgoals: destroying can be brought about by overcoming, which can be brought about by seizing, which can be brought about by capturing. Because the problem CAPTURE-FORTRESS is associated with the newly

---

[3]The radiation problem is how can a ray source be used to destroy a tumour in a patient when full strength radiation will destroy health tissue leading to the patient's death, but less intensity will not destroy the tumour.

[4]In this problem a general in command of an army must capture a fortress, but frontal attack by the whole army is impossible.

activated concept CAPTURE as well as the concept BETWEEN, it accumulates sufficient activation to trigger analogical problem solving."

It is unlike Gentner's model in determining which elements are mapped from source to target. Gentner's model favours the mapping of relations over features, and information that belongs to an interconnected cluster of concepts is especially favoured. PI does not use such syntactic criteria, but instead the subgoal-transfer mechanism ensures that elements of the source problem that proved important in providing a solution are transferred over to the target problem. This means that it has the potential for finding analogies where surface features are not shared. However, it does assume the specific context of problem solving, - and that a solution has been found in the source domain. It is not clear how generaliseable such a model is; for example algorithm transfer is very different in its mapping from an object/relation structure, and so use of the PI model may be restricted to similar domains. With domains that are very different there is the issue of how they can be represented within this system. The main strength of the model is that it begins to address the question of how a source analog is selected, whereas Gentner's model, for example, starts from the point where the analogy has been chosen.

The literature on analogies[5], then, suggests that analogies play an important role in bridging the gap between the known and unknown and can help learners assimilate new information, generate and test hypotheses. Furthermore, according to Carroll and Mack, even if they are not provided they are likely to be generated spontaneously. Of the theories considered, neither operational nor stuctural accounts specify the process by which mapping occurs, although Gentner's model enables various properties of the mapping to be defined and evaluated once the

[5]According to the definition given earlier, some are metaphors and some are analogies

mapping has occurred. Holyoak and Thagard's model begins to address the issue of how the mapping is made, but only within a very specific and limited domain.

3.8 METAPHORS AND ANALOGIES IN TEACHING COMPUTING CONCEPTS

There has been little work on developing theories on the use of analogies in the area of teaching programming, although as we have seen, Carroll and Mack (1983) and Bott (1979) have carried out research in the domains of word processing and text editing. Bott argued that as a learner with partial knowledge will attempt to assimilate all new information into the analogy it is better to provide learners with analogies, rather than let them find their own, because then the teacher can make the mappings explicit and point out the elements that do not map. But as was also pointed out, if metaphors are necessarily open, misunderstandings can still occur, as metaphors are likely to be generated spontaneously. Many teachers of programming and textbook designers have been aware of the problems of novices lacking an appropriate conceptual framewrok to help them learn and have incorporated analogies with everyday domains to help overcome this problem. However, instructors and designers have had few guidelines about what analogies are best to use, and there has been little or no evidence about the effects of using such analogies in teaching. Table 3.1 below lists examples of analogies from two introductory programming texts, and particular metaphors for recursion, from Hofstadter (1980). (These are paraphrased where necessary):

Concept	Analogy
computer, memory, value	a computer is like a drawer: a big drawer for real numbers, a small drawer for integers (drawer has symbolic variable name; name stays with drawer but contents change)
functions	commonly used functions stored as library (illustration of book) in computer
algorithm	algorithm is a recipe (a procedure for solving a problem which quits after a finite time)
argument	an argument of a subroutine is like a two way street; it's to get values into subroutines and to get values out. (functions only lead in).

(Above are all from the "FORTRAN Colouring Book", Kaufmann, 1978)

programming	programming is like playing a musical instrument: you can only learn with practice
program	a program is like a living cell: random mutations are nearly always bad, and usually fatal
conditionals	(analogy is with use in "normal" English) Example: "if the last train has gone, you'll have to spend the night in Aviemore"
variable	the analogy is with two score-boxes labelled with football team names, and starting with zeros in, and you are the score keeper. The memory of the computer is like a large blackboard. When the machine is first switched on, this is wiped completely clean; when a variable is first mentioned, the computer draws a box, and writes a number into the box. (Stores the appropriate value in the memory which has been set aside). The previous value is replaced.

(Above examples from: Introduction to BASIC, part 1: Commodore VIC-20)

Recursion	figure and ground - figure reproduces itself in ground
	story within story within.....
	pictures within pictures - Escher picture of man in gallery looking at picture of man in gallery looking at picture of.....
	nested tales
	telephone calls (holding caller A while speaking to B; holding B while.... then going back up the chain)
	look ahead in chess
	DNA

(from Hofstadter, 1980)

Table 3.1: Analogies and metaphors used in introductory programming texts

These examples include very different kinds of analogies. The intention of comparing programming with playing a musical instrument is presumably to reassure the learner who is struggling that it does take time and application, whereas the analogy with the use of English for the conditional gives some information about how the conditional works, as it is assumed that the learner is familiar with using conditionals in everyday language. The analogy for a variable partly explains which element is mapped from a base domain (the score boxes are equivalent to variables) and describes some properties of variables (that previous values are replaced and it is inferred that the box can only store one value at a time). This particular analogy is very similar to the one used by Mayer (1975) in his concrete model for BASIC, which was discussed in chapter 2, and which was

used successfully to teach novices. Probably most of the work on analogies in the domain of programming has been where they are incorporated into conceptual models for teaching novices where the workings of the notional machine are presented by references to mechanisms with which novices are familiar. The next section looks at one example of the conceptual model approach, and points out some limitations and suggests ways of addressing them.

3.9 THREE WAYS OF DESCRIBING THE CONCEPTUAL MODEL

In chapter two, du Boulay, O'Shea and Monk's work was discussed; in particular their suggestion that a good conceptual model for teaching novices should adhere to the principles of simplicity and visibility. However, whilst these principles can guide the designer in providing a descriptive model which the novice can use to plan and interpret the machine's response, they do not sufficiently address the issue of how the conceptual model is presented. Here I define three further ways of describing the internal structure of the conceptual model: in terms of function, procedure and state[6]. How the conceptual model is presented to the novice will affect which of these aspects is given most emphasis, which in turn will influence the novice's learning experience. The three aspects of the conceptual model described here complement and complete the criteria outlined by du Boulay, O'Shea and Monk, and they provide a tool for analysing conceptual models which is used in chapter 4. They can be defined as follows:

State description The state of the machine at any point in time, the data structure, values of variables, and so on.

---

[6]The idea that the conceptual model could be viewed as having these three distinct aspects was suggested to me by Dr M. Sharples, of Sussex University.

Functional description

A description of the system in terms of the goals of the task and a description of the class of algorithms for attaining that goal. The goal could be expressed at various levels of detail. In a low level language such a goal might be to "add 2 numbers", and the method for achieving it would be described in terms of the contents of particular registers. In a higher level language the goal might be to sort an unordered sequence of numbers into ascending order and the algorithm for achieving this might be a SORT routine. At this level it is close to the plans which programmers use. Some goals might be achieved by a single statement or command, e.g. a Logo repeat statement has the function of repeating commands.

Procedural description

A description of the machine in terms of the set of operators and their pre-conditions for carrying out a task. In Logo examples include the commands for moving the turtle or for printing on the screen. In order to have a complete procedural description for drawing a square, the necessary pre-conditions include the position of the turtle on the screen. In a low level language an example is the command for incrementing the contents of a register.

Thus the functional description is distinguished from the procedural as being a teleological description (describing the problem in terms of goals) rather than operational (describing the problem in terms of a sequence of operations).

These three descriptions can be thought of as different views of the conceptual model, and a conceptual model is unlikely to give equal emphasis to these three different views. Which of the three views is emphasised will depend on a number of factors which include the language, the types of examples given, the style of the instruction and the representation used. These three views of conceptual models apply to any machine or system that can be represented by a state transition network, so it has a start state S^1, and a series of operators or functions which

transform S^1 to a series of subsequent states S^2.....S^n. So O^1 (S^1) => (S^2), O^2 (S^2) => (S^3), or as a state transition network S^1 ----->S^2 ---->S^3 etc.
For each sequence of states there is a start state S^1 and an end state S^n, and at this level one other more general operator O^n which transforms S^1 into S^n: i.e.

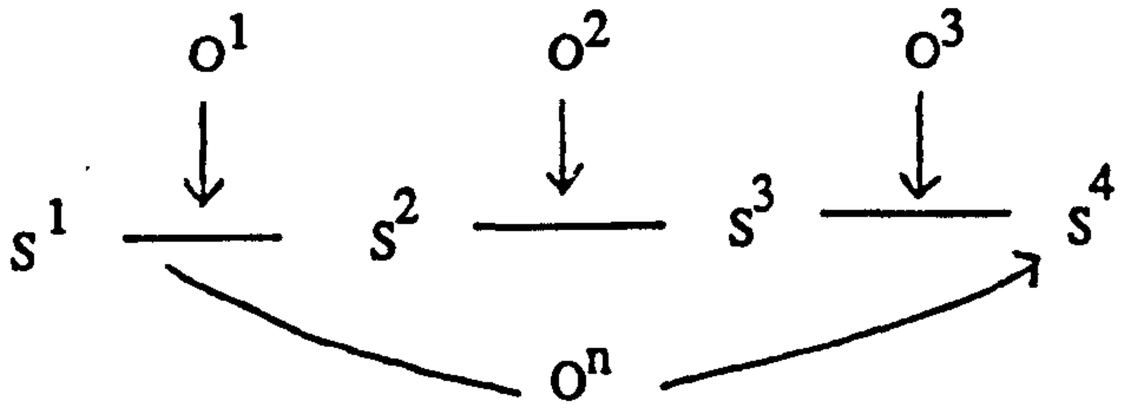


Figure 3.1: How operators transform states

For a sequence of states there is a series of operators acting at successively higher levels of generality. Each of the three views is described in more detail below.

State description

A state description lists each state S^1 - S^n at some level of detail; the level of detail defines the level of the conceptual model. For example, consider a 'SORT' program which sorts a list of letters into alphabetical order by swapping pairs of elements in an array:

S^1	DACB
S^2	ADCB
S^3	ACDB
S^4	ACBD
S^5	ABCD

The columns represent 'registers' and S^1 - S^5 are the time states, so we have a state description showing the state of the machine at time S^1 - S^5 which helps us to understand how the 'swap' occurs - and thus how the function works.

Functional description

A functional description specifies the goal at some level of detail and the functions for achieving that goal. Thus the SORT program has a sub-goal of swapping two characters which can be described as follows:

```
(SWAP CHARACTERS)
 MOV M,D                STORE First character
```

The important thing is that the description is in terms of the goal of the task, (here it is to swap characters). In a functional description of a higher level language, the goal itself might be to sort. If a SORT routine exists, or the programmer knows how to sort, this functional description would be helpful. Such a routine, could, however, be subdivided into smaller tasks: in assembler a functional description of some of these might therefore be:

```
CLEAR SWAP FLAG
SET LOOP COUNTER REGISTER
COMPARE NEXT TWO CHARACTERS
SWAP CHARACTERS
SET FLAG
```

All of these can be described as goals or sub-goals. Another example of a functional description is:

```
LOAD B
 ADD C
 STORE A
```

Functional descriptions often assume unspecified operators, i.e. there is a general class of procedures that can be written to specify the operator rather than a single already defined operator. For example a functional description could be a low level statement to add two numbers. Again this is a goal, a statement of intention; the numbers are to be added together, but note that there is no information about

how it would be done. In fact, the structured programming approach advocates building up a functional representation with the details getting progressively filled in. The flow charts in figs 3.2 and 3.3 below are examples of this.

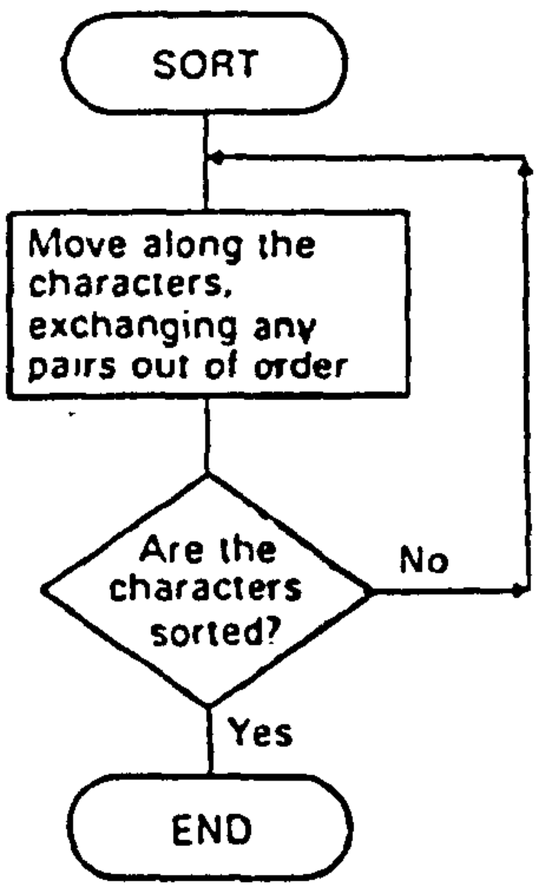


Figure 3.2: Flow chart of SWAP routine

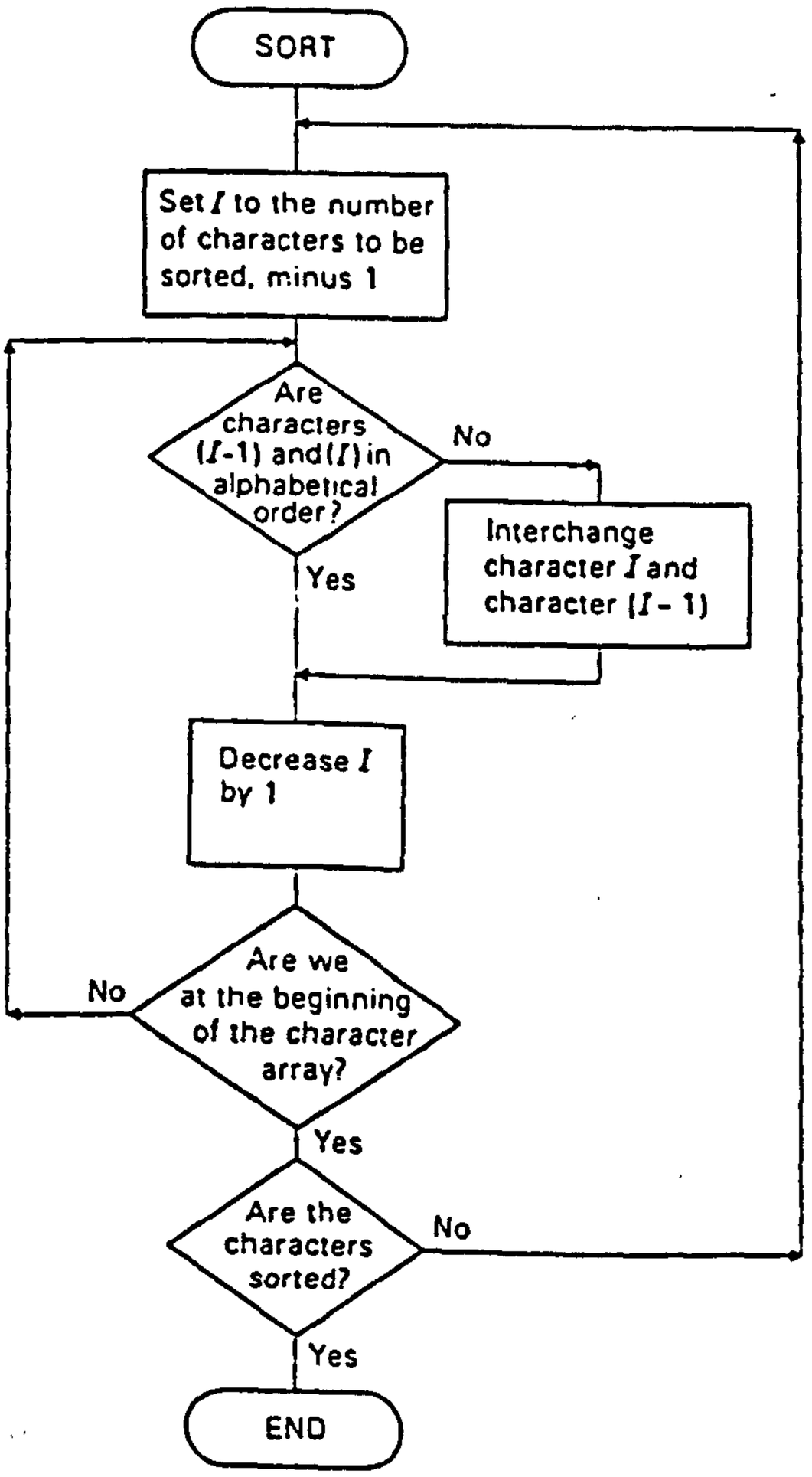


Figure 3.3: Flow chart of SWAP routine

A functional description is similar to the programming plans discussed in chapter 2

(e.g. Soloway et al. 1982) although it won't contain information on when the plan can be used and how major parts of the plan can be interleaved within the rest of the code. It is likely, therefore, that a functional view will facilitate the development of plans. This point will be taken up again in chapter 4.

Procedural description

In a procedural description the operations for carrying out the task need to be fully specified, e.g.:

BUILD 'PAYROLL

READIN DATA

assumes that algorithms have been defined for these sub operators (sub procedures). According to this definition, the information in the flow charts in figures 3.2 and 3.3 could be made procedural by giving a start state (i.e. the contents of the registers). The SORT routine assumes a read-in routine (of some kind) and in order to carry out SORT we would need to know where the characters are stored. Given this information, and a start state, we would have a procedural description. However, the emphasis is now on describing the problem in terms of a sequence of operations rather than the goals to be acheived. The start state in the SWAP example is:

S^1 D A C B

The operations which act on the start state also need specifying at some level of detail. The instructions given earlier for the functional description :

CLEAR SWAP FLAG

SET LOOP COUNTER REGISTER etc.

in conjunction with the start state, provide a procedural view.

Here is another example from a different domain:

Ann at MK Station

BR train MK Station --> Bangor Station

Ann at Bangor Station etc

So there is a hierarchy of sorts in that a procedural description can be composed from a state description plus a functional description. However, procedural descriptions and functional descriptions may be couched at different conceptual levels, e.g. the **functional** view may be

take transport house -> Bangor (i.e. the goal is to move from the house to Bangor);

whilst the **procedural** may be

Ann at House

Ann take taxi house -> MK central station

Ann take train MK central -> Rugby etc

Thus a functional description may seem more powerful because it is at a higher conceptual level and so more general. By more general, is meant that a less general operator is subsumed by another, thus "take train" is less general than "take transport". So a functional description at a high level allows many different procedural descriptions at lower levels because it focusses on the goal.

There may well be examples where it is difficult to decide which view is being given. Distinguishing a procedural description from a functional description is harder than distinguishing a state description from the other two. For example a functional model could be sufficiently determined for it to effectively also provide a procedural description, giving enough information for us to carry out the necessary operations, but this is not necessarily the case. More than one view may be prominent. This is not a serious problem: what is important is which aspect is

emphasised: for example a functional description emphasises the function or role of the code whereas a procedural description gives the steps for carrying it out. The usefulness of the categorisation is in identifying the different kinds of knowledge which is necessary to understand the conceptual machine. It provides a way of assessing whether the conceptual model offered gives these different views, or whether it is emphasising only one or two views. For example a conceptual model giving only a state description will not be helpful because the algorithm is not fully specified - though listing the intermediate states when giving a procedural model may be helpful since it gives the reader confirmation that her mental model is correct. How this categorisation applies to the languages used in this study will be discussed in chapter 4.

3.10 CONCLUSIONS

This chapter provides a theoretical framework for the thesis. Using conceptual models for teaching novices programming assumes a theory of learning which is broadly Piagetian: the novice, precisely because she is a novice, cannot easily build up the cognitive structures needed for assimilating the new knowledge she is encountering in learning programming. Conceptual models aim to provide a knowledge structure into which the learner can assimilate her learning. This Piagetian framework is compatible with an Artificial Intelligence approach: both are structural approaches, and the similarities and links between them are further explored by O'Shea and Self (1983). Sections 3.4 to 3.8 considered the role of analogy and metaphor, and it was argued that they help the novice to make connections between existing knowledge and new knowledge, and for this reason, analogies are often used in conceptual models. The terminology used in this area of mental models was clarified, and a categorisation of the different aspects of conceptual models was offered. The next chapter will show that the conceptual models provided often do not give equal weight to the different aspects, and will discuss the relevance of this for acquiring programming plans.

Chapter 4

INTRODUCTION TO THE PROGRAMMING LANGUAGES

Contents **Page**

4.1 INTRODUCTION

Chapter three discussed and criticized the principles offered by du Boulay, O'Shea and Monk (1981) for designing conceptual models for novices, and also identified three further ways of describing the conceptual model which distinguished between state, procedural and functional descriptions. In this chapter the languages and curricula which students learnt in the study are described, and the conceptual model associated with each language is analysed and discussed: both using du Boulay et al's criteria, and secondly using the different descriptions of the conceptual model identified in chapter three.

Four languages were used in this study: SOLO, the PT501 assembler, Logo and the DESMOND assembler. The PT501 and DESMOND assemblers will be referred to as PT501 and DESMOND respectively. Each of the above will be treated in turn, and described in order to help the reader to follow the examples which are given in chapters 5, 6, 7 and 8. All of these languages are part of larger programming environments which include items such as error messages, tracers, programming tutorial manuals, experiment books and so on. Each is presented to the novice via a *conceptual model.* Prescriptions suggested by du Boulay, O'Shea and Monk (1981) for designing such conceptual models were discussed in chapter 2. In this chapter, the four environments, and in particular the conceptual models, will be analysed and discussed using two different frameworks: the first is based on the prescriptions given by du Boulay , O'Shea and Monk (op. cit.) which were discussed in chapter 2, and the second is the categorisation of conceptual models offered in chapter 3. This evaluation is summarised as a "WHICH" guide to the four environments.

One addition will be made to the criteria of simplicity and visibility offered by du Boulay and O'Shea. It was pointed out in chapter 2 that some of what was

discussed under simplicity and visibility is closer to what can be termed consistency. Consistency is the extent to which the behaviour of various parts of the conceptual machine is related. An example of this would be whether a particular command can be inferred from another. In Logo, having learnt the LEFT instruction, one can reasonably infer that there will be a RIGHT instruction. This principle applies in environments other than programming, and may even be more important in such applications in that if there is consistency then one application can be used by analogy with another. The Apple Macintosh is a good example of this principle, and enables users to transfer between applications. For example a user can assume with an unknown piece of software that she can take the mouse to one of the headings at the top, many of which stay constant across applications, click to obtain the menu, and drag the mouse down to where she wants and release to activate the appropriate action. In other words using a new application is done by analogy with the old, known application. There is also another more general type of consistency, which is whether the conceptual model is always presented in a consistent way. The conceptual model is not solely presented through one channel, such as the instructional text, but through error messages, what appears on the screen, etc. All these should be consistent.

4.2 SOLO

SOLO was designed to introduce cognitive psychology students to Artificial Intelligence as a tool for modelling human cognition. It was devised by Eisenstadt (Eisenstadt, 1978) who describes it as a "friendly software environment" (Eisenstadt, 1979). The intention was that SOLO should be a system which makes it as easy as possible for the user to get the system to do what she wants it to without the user getting tangled up because of trivial spelling and syntactic errors. In order to do this, it should trap such errors thus optimizing productive interaction. The idea is that "virtuous" or constructive bugs would be encouraged, maximising

the interesting conceptual errors that are part of the learning process. Its main component is a language for manipulating a relational data-base, containing facilities for inserting descriptions into the data-base and for pattern matching against descriptions already in the data-base.

The constraints (Eisenstadt, op. cit.) were that it should introduce students to A.I. in a short time (apart from a 2 day summer school project, 2 weeks part-time study and 6 terminal hours were allocated), cover the fundamentals of programming and be accessible to students with no computing background who may be antagonistic to computing. Additionally, it should motivate students, be enjoyable and of direct relevance to their interests as psychologists. At the time of its inception, another constraint was that it would be used at hard copy terminals rather than VDU's, and apart from summer school access, access would be through one of 160 terminals at Open University study centres. The manual is designed to persuade people to work in a structured manner: for example a restriction is set on the length of the procedures, so that programming problems may involve using several smaller procedures for each section of the problem.

The language has two main elements: the data-base, and the procedures which act on the data-base. The data-base consists entirely of triples, which take the form node-relation-node. Below is a simple example of how the data-base might look:

```
DOGS--->EAT--->MEAT
|
|----->HAVE--->FLEAS

FIDO--->ISA---> DOG
```

The right pointing arrow between the first node and the relation, and between the relation and the second node, is always there whenever the triple is printed out on a terminal or screen, and serves as a reminder that the relationship between the

elements of the triple is unidirectional.

There are 10 primitives which act on the data-base to access, add and delete triples and so forth. These are the SOLO built-in procedures: BYE, CHECK, DESCRIBE, EDIT, FOR EACH, FORGET, LIST, NOTE, PRINT and TO.

NOTE, FORGET and DESCRIBE

Triples are added to the data-base by the use of NOTE, e.g.

NOTE DOGS HAVE HAIR

would result in the data-base above being updated to read:

```
DOGS--->EAT--->MEAT
|
|----->HAVE--->FLEAS
|
|----->HAVE--->HAIR

FIDO--->ISA---> DOG
```

All the triples starting with a particular node can be accessed by using the instruction DESCRIBE (node). Thus DESCRIBE FIDO would result in

```
FIDO--->ISA---> DOG
```

appearing on the screen. Deleting a triple from the data-base is achieved by using the primitive FORGET, e.g.

```
FORGET DOGS EAT MEAT
```

which would result in the data-base looking like this:

```
DOGS--->HAVE--->FLEAS
|
|----->HAVE--->HAIR

FIDO--->ISA---> DOG
```

The CHECK procedure

The CHECK procedure is used to retrieve triples from the data-base, or CHECK that triples are in the data-base, and also, within procedures to implement conditionals. E.g. given the above data-base, typing in:

CHECK DOGS HAVE HAIR

would result in:

PRESENT

Defining a procedure (using TO) and the PRINT procedure

New procedures are defined by using the TO procedure, using the format:
TO procedure (/parameter/). When CHECK is used within a procedure, the user is automatically given the segment of the next line "If present", and is also required to specify the flow of control, i.e. to type in CONTINUE or EXIT. A trivial example should make this clear:

TO CHECKDOGS
10 CHECK DOGS HAVE HAIR
10A If Present: PRINT "YES, DOGS HAVE HAIR"; CONTINUE
10B If Absent: PRINT "NOT AS FAR AS I KNOW":CONTINUE
20 CHECK DOGS HAVE FLEAS
20A If Present: PRINT "YES, THEY HAVE FLEAS"; EXIT
20B If Absent: PRINT "NOT AS FAR AS I KNOW":CONTINUE
DONE

Figure 4.1: Automatic prompting for conditionals

The bold segments appear automatically, and the underlined segments, the control statements, must be typed in before the user can go any further.

This example also shows the use of the PRINT procedure. There are also two flow of control statements: CONTINUE and EXIT. The sub-lines 10A, 10B, 20A and 20B appear automatically, and the programmer must give a flow of control statement: either CONTINUE which takes control to the next instruction, here it is line 20, or EXIT which terminates the procedure. Procedures are changed using

the procedure EDIT (EDIT procedure) and listed by using LIST, (LIST procedure).

Parameters, variables and making inferences

As SOLO was designed to be a tool for modelling cognition, many of the procedures are intended to model the process of inference. One of the procedures which will be looked at again in chapter is the WEAKASSESS procedure, which takes one parameter. Here is the first part of it:

```
TO WEAKASSESS /X/
1 CHECK /X/ PLAYS SQUASH
        1A IF PRESENT: PRINT "FIT"; EXIT
        1B IF ABSENT:  CONTINUE
2 CHECK /X/ RIDES BICYCLES
        2A IF PRESENT: PRINT "FIT"; EXIT
        2B IF ABSENT: CONTINUE
        ..............
4 PRINT "UNFIT"
DONE
```

The procedure is intended to represent the following inference: if some person (X) both plays squash and rides bicycles, then she is fit. It could be extended to include any games that can be played in the fitness assessment. In this case the second line would be altered to use a variable. Wild-card matching is only allowed on the second node in the node-relation-node triple. The symbol used is a question mark "?" If there is a pattern in the data base which matches, the value of the right hand node is bound to the symbol *. To use a variable in the above procedure, it could be altered as follows:

```
TO WEAKASSESS /X/
1 CHECK /X/ PLAYS ?
    1A IF PRESENT: PRINT "FIT"; EXIT
    1B IF ABSENT:  CONTINUE
2 CHECK /X/ RIDES BICYCLES
    2A IF PRESENT: PRINT "FIT"; EXIT
    2B IF ABSENT: CONTINUE
.............
4 PRINT "UNFIT"
DONE
```

Often the programmer will want to refer to or use the value of a variable later in the procedure, or in another procedure. Letters can be added to the '*' symbol to identify and keep track of variables. Below is another procedures given in the SOLO manual, which shows how inferences can be propagated through a database:

```
TO INFECT /X/
1 NOTE /X/ HAS FLU
2 CHECK /X/ KISSES ?
    2A IF PRESENT: NOTE * HAS FLU
    2B IF ABSENT:  EXIT
DONE
```

This example can be worked through with the data base given below:

```
LIZ
|
|--->KISSES--->HENRY
|
|--->KISSES--->JOHN
|
|--->KISSES--->TONY

JOE
|
|--->KISSES--->JANE

JANE
|
--->KISSES HENRY
```

```
HENRY
|
|-->KISSES JANET
```

If the following line is typed in:

```
INFECT  LIZ
```

the data base will be altered as follows:

```
LIZ
|
|--->KISSES--->HENRY
|
|--->KISSES--->JOHN
|
|--->KISSES--->TONY
|
|--->HAS FLU

JOE
|
|--->KISSES--->JANE

JANE
|--->KISSES HENRY

HENRY
|
|--->KISSES JANET
|
|--->HAS   FLU
```

Recursion

The example above is used to introduce students to the notion of recursion, by changing the procedure above to the recursive version given below:

```
TO INFECT /X/
1 NOTE /X/ HAS FLU
2 CHECK /X/ KISSES ?
    2A IF PRESENT: INFECT *; EXIT
    2B IF ABSENT: EXIT
DONE
```

Both versions of the procedure capture the notion that if a person has flu, and that person kisses someone else, then that second person will also catch flu. However, in the first version, the first line, NOTE /X/ HAS FLU will match the node containing the parameter in the title line, (LIZ in the above example) and put LIZ HAS FLU into the database. Line 2, CHECK /X/ KISSES ? matches the triple where /X/ (also LIZ), is the first node, the value of * becomes HENRY, and the result of line 2A is to put HENRY HAS FLU into the database. In the second, recursive, version the infection is spread to all nodes along a chain of 'KISSES' relations. HENRY KISSES JANET is in the database and so the database would be updated to include JANET HAS FLU, but there is no triple which matches JANET KISSES ? , so no other alterations would be made.

Iteration: the FOR EACH CASE OF procedure

FOR EACH CASE OF is used to cycle through multiple relations. For example, if the following procedure is run with LIZ as the value of the parameter, using the database above, it will match LIZ KISSES HENRY, LIZ KISSES JOHN, and LIZ KISSES TONY and will print out each of these triples.

```
TO KISSCHECK /X/
1 FOR EACH CASE OF /X/ KISSES ?
    IA PRINT /X/ KISSES *
DONE
```

FOR EACH can be used as a sub step within another FOR EACH. The following program is an example:

```
TO CRAVETEST /X/
1  FOR EACH CASE OF /X/ DRINKS ?A
     1A:  FOR EACH CASE OF *A CONTAINS ?B
          1AA NOTE /X/ CRAVES *B
DONE
```

Finally, BYE takes the user out of the SOLO environment.

4.3 SOLO'S CONCEPTUAL MODEL: SIMPLICITY, VISIBILITY AND CONSISTENCY

Simplicity

SOLO rates very well on this criterion. It has a small instruction set (10 instructions) which is reproduced as a summary on the back cover of the programming manual. It is clearly suited to its job of modelling cognition and introducing Artificial Intelligence, and enables students to do interesting tasks (for example modelling children's two column subtraction), but it would not be suitable for number handling. Syntactic simplicity is increased by the fact that when a user types the CHECK instruction within a procedure, she is automatically given the IF PRESENT statement (the THEN part of the conditional), and is forced to specify the flow of control statement (CONTINUE or EXIT) after which the ELSE branch, IF ABSENT, is also given automatically, and again the flow of control statements must be specified. This was shown earlier in figure 4.1. SOLO's documentation (forming two units of the course) has a clear metaphorical description of the SOLO machine, and error messages reflect this. Concepts are introduced by analogy with everyday objects or ideas with which users can be expected to be familiar. For example

Concept	Analogy
Database (SOLO)	Automated telephone directory

Activating a procedure	Procedure is an 'expert' who 'goes into action'.
Procedures with parameters	Procedures are experts and parameters are their own private containers
Flow of control	Passing of batons in relay race
Variable	Repository

There is a potential problem with the repository analogy in that a repository can have a number of contents - although in the diagram it is clear that only one variable is allowed in the repository and the previous contents are lost. These analogies are carefully carried through the teaching manual. For example, when recursion is first introduced the following description is given:

"If it seems somewhat strange that a procedure can activate itself, just keep in mind the following analogy: when a procedure is defined originally, a large 'reserve pool' of procedures is created, like a reserve of football players, sitting on a bench and waiting to go into action. Every member of a given 'reserve pool' behaves like every other member of that same pool-i.e. they are 'experts' at the same task (namely carrying out the definition of the procedure step by step). Suppose that an INFECT 'expert' is busy going through its definition step by step. When it comes across the name of some other 'expert', say NOTE, it simply puts a NOTE 'expert' into action (by 'calling it up' from the 'reserve pool' of INFECT 'experts'). As far as any given 'expert' is concerned, it is equally easy to 'call up' another 'expert' from either its own 'reserve pool' or from some other 'reserve pool'.
An analogy used in section 6 for 'flow of control' was that of passing the baton from one runner to another in a relay race. The same analogy is relevant here, for when any 'expert' calls another expert into action, that same baton is passed on from the first expert to the second (SOLO has many 'experts' but only one baton!). This baton is only on temporary loan, however, and an 'expert' must return the baton to whomever he got it from before 'retiring' to the 'reserve pool'!"

By "conceptually simple" is meant giving enough information to explain what's going on - in a way novices will understand, but which avoids confusing the user with detail which is not yet required. At the beginning of using SOLO the user is told that FIDO can only like or have one thing at a time. This restriction is later lifted. The argument here is for a clearly defined and restricted set of features for the beginner user which can be added to as she gets to know the system. The proficient user is allowed to see more of the system but the beginner is protected

from her own mistakes, and as her skills grow so the notional machine or model grows with her.

Visibility

Some of the ways in which the SOLO machine could be more visible require a screen presentation, and are therefore not possible as SOLO was first intended for access via teletypes. There are various ways, however, of increasing visibility. For example, the data base is shown on the teletype in the same format as it appears in the manual (see fig 4.2) and the 'arrow' link between a node and its relation and the relation and the next node emphasises the unidirectional nature of the relationship.

```
HENRY
|
|--->KISSES->JANET
|
|--->HAS->FLU
```

Figure 4. 2: Format in which the SOLO data base is presented in manual and on screen.

The analogy is drawn between this and the unidirectional nature of a telephone directory. However, in the early versions of SOLO, some of SOLO's facilities conflicted with the notion of visibility. For example, one aspect of visibility is informing the users of any changes made, and unless the facility is suppressed SOLO informs the user of any changes made in its database. If however, a user declares a parameter in the title line of a procedure, and then fails to refer to it in the body of the procedure, SOLO assumes this is an error and deletes the parameter in the title line without informing the user. SOLO's spelling checker can also be over zealous, and this can lead to the kind of trivial problems that it was intended to avoid. Suppose that, with a FIDO triple in the database, the user types in NOTE FIGS HAVE LEAVES, early versions of SOLO with its over-zealous checker would respond "WHEN YOU TYPED FIGS DID YOU MEAN FIDO?"

Consistency

Some areas of SOLO are not consistent in the sense of being able to infer the instruction to be used in one environment from another. For example there is no equivalent of FORGET (which is applied to nodes) for procedures. (There is in fact a procedure for deleting procedures but the user is not told this). Also the database is DESCRIBEd to give a list of nodes, while procedures are LISTed. Apart from these instances, SOLO rates quite well on consistency. As was shown earlier, the database is always shown in the same format whether it is in the manual or on the screen or teletype and error messages are also consistent with the manual.

A "WHICH" type star rating will be applied to each of the four environments which will be rated on simplicity, visibility and consistency. Although this is obviously somewhat subjective, it does provide a means of comparison. Using the star system[1], SOLO's rating is:

Simplicity	****
Visibility	**
Consistency	***

---

[1]Key to star rating;

****	Very good
***	Good
**	Moderate
*	Rather poor

4.4 SOLO'S CONCEPTUAL MODEL: STATE, PROCEDURE AND FUNCTION

In this section SOLO's conceptual model is examined in terms of the additional categories, discussed in chapter 3, of state, procedure and function.

Whenever changes are made to the data base, the up-dated data base is automatically printed. This is a state description, (see fig 4.2, given earlier) as it is showing an aspect of the state of the machine at a point in time, in this case the aspect shown is the data-base. There is also some emphasis on a procedural description, in that users are shown, in the manual, sequences of commands to achieve a task. The analogy for variables given in the manual (contents of a box) highlights the procedural view, and for parameters and procedures (expert workers with instructions and slots) the emphasis is also procedural. Both analogies are showing how objects are moved around within the machine.

One view which is not emphasised is functional, except at a detailed level. For example there is a detailed account of "how complicated sequences of CHECKs within CHECKs can be achieved in a single procedure by arranging your EXITs and CONTINUEs properly". The examples used are the STRONGASSESS and WEAKASSESS procedures, and these illustrate the function of the CONTINUE and EXIT statements. These procedures are also examples of how to achieve OR and AND decisions, which can be seen as a higher level functional view; but they are not described in this way. Hasemer (1983) identified 9 SOLO plans (which he calls clichés) but these are never labelled as such in the manual. SOLO's database structure leads to a strong data/program distinction and emphasis on a state description (showing the changes to the database). There is also a procedural view with the emphasis on flow of control but there is little emphasis on the functional view. For example, although the explanations and descriptions of iteration and

recursion are very clear, there is no labelling of the sections of code that achieve these functions - so that the example programs are implicit functional models (e.g. "flu test" as recursion). It is not sensible to try to do a "WHICH" rating on the different aspects in the same way as was done for the criteria, as they reflect emphases, but it is possible to say that there is a strong emphasis on the state of the machine, slightly less emphasis on the procedural description, and hardly any functional description.

4.5 SUMMARY

SOLO fits very well with the criterion of simplicity, having a small instruction set and being well suited to its job. Its documentation gives a clear metaphorical description which is reflected in error messages and the analogies used are carried through. The restrictions which suit beginners are lifted as the learner becomes more proficient. Visibility is more limited as SOLO was designed for access via teletypes. Because of the machine's simplicity, however, this presents less of a problem, for example SOLO's control structure forces a certain amount of procedurality, and so it is less easy to produce large unstructured procedures (though it is quite possible to pass control to and from procedures and lose track of what's happening). Early versions of SOLO had features which were not consistent with the criteria of consistency and visibility, but these have been changed in later versions. Versions of SOLO have also been produced (e.g. Hasemer, 1983) which enhance visibility by giving the user tracing facilities, and put more emphasis on a procedural description. An analysis of SOLO in terms of the different aspects of its conceptual models indicates that the emphasis is on procedural and state descriptions but that there is little higher-level functional description.

4.6 OPEN LOGO

Unlike the other programming languages, Logo is part of a philosophy of teaching (Papert, 1980), and it is necessary to consider Logo in this context, as it affects how Open Logo is taught. For example, the teaching approach adopted in the OL tutorial manual is somewhat different to the approach adopted in the other curricula: more personal and exploratory. For this reason, the philosophy will be discussed here, and the curriculum will be discussed in chapter 8 which reports on the study of Logo.

The language and philosophy

Logo can be viewed on two levels: the first is Logo as a programming language and the second is Logo as a tool for thinking, for problem solving.

Logo as a programming language

Like SOLO, Logo is modular, i.e. programming projects are subdivided into small pieces and a separate procedure is written for each piece. It is also recursive (again like SOLO). A procedure can be written which can call itself. Unlike SOLO, Logo is extensible. This means that procedures defined by the user look and behave like primitives or built in operations. For example most versions of Logo do not have primitives for looping such as FOR, DO or WHILE, but these can be defined to look like the Logo IF primitive.

Logo as a tool for learning

Logo was designed to be used by people with no knowledge of maths or computing, but to be powerful and provide an interesting entry point to mathematics. This entry point is, of course, turtle geometry. However, the grander claims for Logo have been much more far-reaching and can be summed up in the following statement:

"Logo's goal is different..... It isn't supposed to be an easy introduction to something else..... and it isn't a tool for teaching the same math curriculum people are already teaching. Instead it's a door into the territory of the computer as an object for intellectual exploration..... Logo is for learning learning" (Harvey, 1984)

As discussed in chapter 2, some of the claims made for Logo are that it can teach people to reason, problem solve and plan, or rather, help people to learn these things for themselves. For Logo carries with it a philosophy about how it should be taught. The best exposition of this philosophy is given by Papert (1980) in his book "Mindstorms", where he says:

"In my vision the child programs the computer, and in doing so, both acquires a sense of mastery over a piece of the most modern and powerful technology and establishes an intimate contact with some of the deepest ideas from science, from mathematics, and from the art of intellectual model building" [Papert, 1980].

This philosophy assumes an open approach to learning where the sequence of concepts learnt will depend largely on the learner, and the judgment of the teacher. For distance teaching, this raises the issue of how such an approach can be packaged.

A powerful effect of any programming language (which proponents of Logo are particularly keen to exploit) is that the outcome of our own thinking, the program, can be tried out (run), and, so the argument goes, diagnosing the bugs in the program can correspond to diagnosing the bugs in our thinking. However, for this approach to work it is necessary for bugs to be viewed as positive rather than negative and so an exploratory approach is often advocated which encourages the learner to view bugs as creative. For Logo to produce the results it might be

capable of, it needs to be embodied within the structure of a very well conceived curriculum - and individual learners need guidance to the right point (and of the right amount) to make their discoveries.

Open Logo is part of a package which includes the Open Logo chips, the reference manual and the tutorial manual. In this study students only had access to the tutorial manual. Logo is a LISP-like language. The total instruction set is quite large: 90 instructions are given in the OL tutorial manual. However, only a relatively small subset is needed for most of the work, which is turtle graphics. 21 instructions form the majority of those used, and are described in the order in which they appear in the manual, in appendix 4.1. The examples given in table 4.1 below, however, are typical.

FORWARD x	Moves the turtle x units in the direction in which turtle is facing, and (in default mode) leaves trace on screen
RIGHT x	Turns turtle position x degrees right (i.e. clockwise)
REPEAT x[ ]	Carries out the instruction in the brackets x times

Table 4.1: Typical Open Logo instructions

Figure 4.3 below gives an example of a simple Open Logo procedure which produces a hexagon.

```
TO HEX
 REPEAT 6 [FD 50 RT 60]
```

Figure 4.3: Open Logo procedure for drawing a hexagon

4.7 LOGO'S CONCEPTUAL MODEL: SIMPLICITY, VISIBILITY AND CONSISTENCY

Simplicity

Of the 22 instructions given in appendix 4.1, a small number can be termed programming instructions. If PENUP and PENDOWN are excluded then there are only 7 programming instructions: - PRINT, FORWARD, RIGHT, LEFT, BACKWARD, REPEAT, TO. That is, they are all instructions frequently used to create a procedure or within a procedure, as opposed to instructions that are concerned with changing the environment or changing mode. Thus the subset of instructions dealing with the turtle can be seen to adhere to the notion of simplicity: indeed this is one of the examples cited by du Boulay, O'Shea and Monk, who say that such a subset : "*initiate a rich and interesting but delineated set of actions*".
The number of graphics related commands (those changing the environment such as CLEAN, WIPE etc) and some not mentioned here such as PAINT increase the instruction set considerably.

However, Open Logo's syntax for defining a procedure, and calling a procedure does not distinguish between the two, which, according to du Boulay, O'Shea and Monk, violates the simplicity criterion. Another non-adherence to simplicity in this version of Logo is its error messages. The error messages, along with any other commentary:*"should be couched at a level of detail appropriate to the novice's task and to his understanding of the underlying concepts"*. Consider, however, the two following examples of typical Open Logo error messages:

"Out of space in procedure X"

"BOX has no meaning in procedure X"

The second message gives some idea of the underlying cause: there is no value

available for BOX; in the first, however, the novice is unlikely to know why the procedure has run out of space: in fact the most common reason is writing an accidentally recursive procedure, with no stopping condition, but more of that in chapter 9. The point here is that the message is not given in terms that the novice can understand. The best (or worst) of all, is the message that appears and says:

"Please switch the machine off and on again"

The error messages are part of the environment: part of the conceptual model which is given to the learner. Open Logo therefore rates much lower on simplicity than it would if it had better error messages.

Visibility

When using turtle graphics on fairly simple programs, Logo does well, as the effect of each instruction is visible: the turtle leaves a trace. However this does not allow the learner to readily trace the flow of control in the recursive programs introduced, as the trace is drawn too quickly, although there is a 'walk' facility for stepping through program execution. It is not apparent what mode the machine is in, nor the significance of the different modes, though it is clear (from the screen) when one is in a text mode as opposed to a drawing mode.

Consistency

Most of the Open Logo instructions have internal consistency in that 'opposite' commands can be predicted. For example, it can be predicted that along with FORWARD, LEFT, PENUP and ARCL there will also be BACKWARD, RIGHT, PENDOWN and ARCR which achieve the opposite effects. The error messages, discussed above as an example of lack of simplicity, are also inconsistent, in that the Logo machine is discussed at a different level, and in different terminology to that used in the instruction manual. Logo's star rating is:

Simplicity	**
Visibility	***
Consistency	**

4.8 OPEN LOGO'S CONCEPTUAL MODEL: STATE, PROCEDURE AND FUNCTION

The conceptual model for Logo is not as explicit as the conceptual models for the other languages described. The user is invited to type things and told what the result will be - which is by default a procedural description. Few analogies are given. Some examples of the different aspects which are highlighted are given below. The first example in figure 4.4 gives the text from the tutorial manual and a representation of the screen showing the turtle and its trace within the screen, and the instruction which produced this trace on the right, just outside the "screen":

As it moves, the turtle draws a line to record where it has been

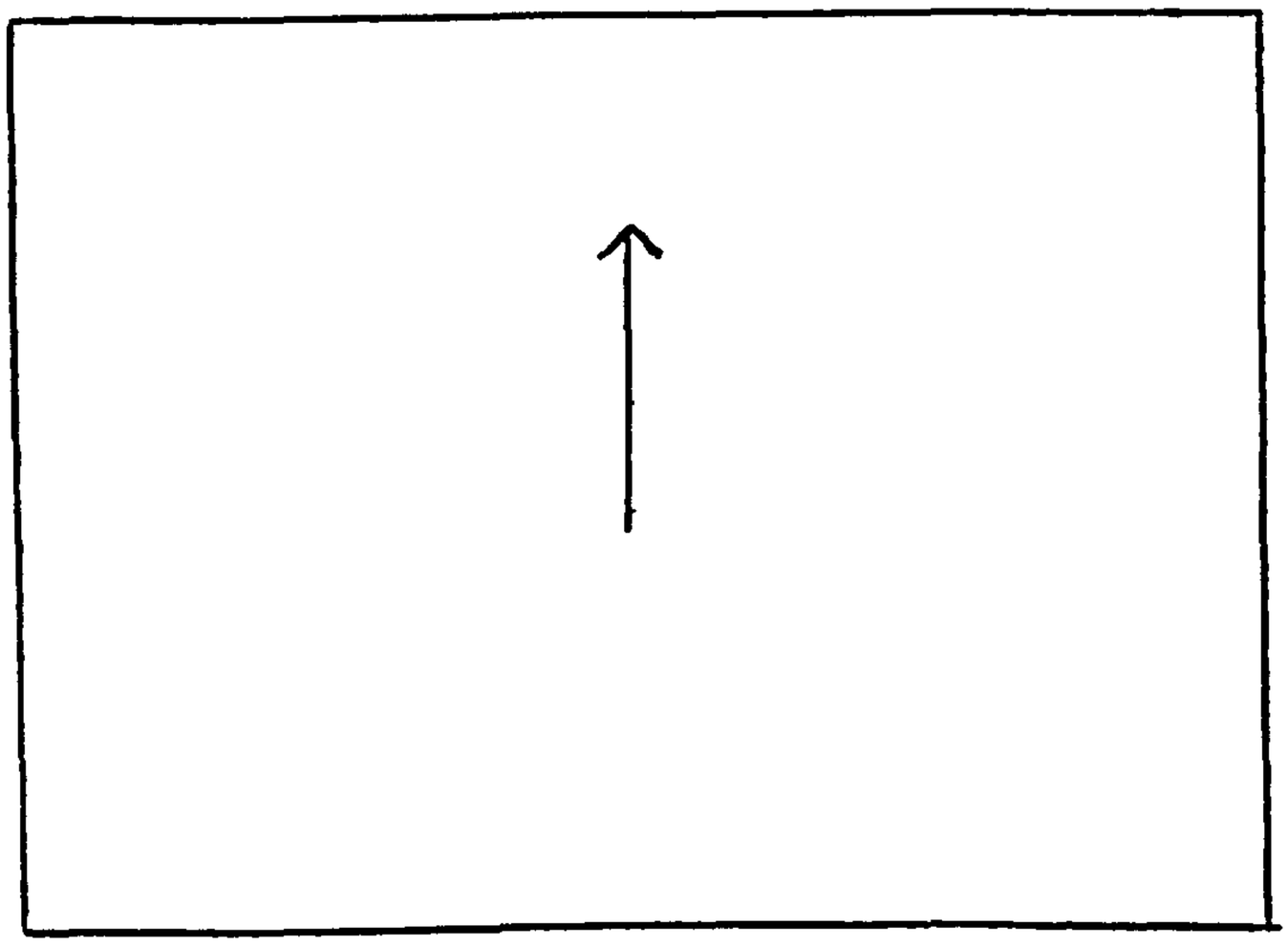


Figure 4.4 How to draw a line

This is closest to a state description: the trace of the turtle reflects a change in the state of the machine. In a slightly more complex example, information can also be inferred about the value of variables. In the instructions for drawing a square

given below, the emphasis is on function: a square will only be drawn if the starting point of the turtle has an orientation that is vertical or horizontal with respect to the screen, so this is not a procedural description, because the starting state is not given:

You type FD 200 RT 90 BD 200 LT 90 BD 200 RT 90 FD 200 RT 90

Figure 4.5: How to draw a square

The instructions given for drawing a triangle are given below. As with the instructions for a square, the emphasis is functional rather than procedural, although both only just fall short of being procedural, and the detailed step-by-step instructions certainly have a procedural feel:

You can draw a triangle using these instructions

REPEAT 3[FD 200 RT 120]

Early sections of the manual are concerned with observing the effects of particular instructions on the turtle. The learner is told that:

The instruction LEFT 45 turns the turtle half left (that is it points it 45 degrees to the left of where it was pointing before it received the instruction).

A screen diagram shows the effect of moving LEFT 45. In concentrating on the commands for moving the turtle, or printing on the screen, the emphasis is on a procedural view. Very early on in the manual, a detailed sequence of instructions is given, and the user is invited to predict the outcome:

FORWARD 100
RIGHT 90
PEN 2

```
BACKWARD 50
LEFT 45
PENUP
FORWARD
```

Again this is a mainly procedural view, although it could be argued that there is also a functional representation in that the learner is being asked to think about the function of each instruction, but as with SOLO, the functional views are at a low detailed level. Most of the emphasis is on the state and procedural descriptions with much less on the functional descriptions.

4.9 SUMMARY

Logo generally scores well on the criteria suggested by du Boulay, O'Shea and Monk: in simplicity due to the restricted command set of turtle graphics, in visibility also in respect of turtle graphics and consistency in terms of the instructions. However, Open Logo, as presented through the tutorial manual, is weak in a number of ways. These include the fact that the syntax does not distinguish between defining and calling a procedure, the inaccessibility of the error messages and the confusing number and nature of the modes. At times the tutorial manual fails to address novices and assumptions are made about their mathematical skills. However, this last point is about instructional design, which will be discussed in chapter 8. The next section will briefly compare the two high level languages, SOLO and Open Logo, in terms of their conceptual models.

4.10 A COMPARISON OF SOLO AND OPEN LOGO

Du Boulay, O'Shea and Monk discuss how computing concepts can be introduced to novices via analogies with familiar objects or ideas. SOLO and Open Logo

differ greatly in the use made of such analogies in the programming manuals: the SOLO manual has 5 or 6 analogies which are carried right through, whereas the Open Logo tutorial manual makes very little use of analogy.

In terms of the instruction set and matching the learner's needs, both Open Logo and SOLO have a limited instruction set if only turtle geometry is considered, and both are suitable for the job in hand. However, outside the turtle geometry subset, Open Logo has a much larger instruction set, and is more powerful. Unlike SOLO, there are no syntactic restrictions, such as forcing the user to specify the ELSE branch of the conditional, nor is there forced modularity; whereas in SOLO, the use of a CHECK within another CHECK is illegal, and so the user is forced to define a procedure to achieve the same result. Open Logo's turtle geometry has visibility, in that the movements of the turtle on the screen reflect changes in the state of the machine, and in very simple programs the effect of each instruction is apparent; but as programs get more complex, the turtle's movement is too fast to follow.

The biggest difference for the novice, however, is in the success of matching the curriculum to the learner. SOLO's instruction manual is extremely good, and no unwarranted assumptions are made about the learner's skills or knowledge. Open Logo is much weaker here, and provides unhelpful error messages and assumes prior knowledge in the maths examples. Both emphasise procedural and state descriptions at the expense of functional descriptions, and both aim to match the learner's needs, but SOLO is the most successful in achieving this aim.

4.11 PT501: THE INTEL 8049 ASSEMBLER

This will be referred to as PT501, the code number of the Open University course which includes it. The assembler is a version of the Intel 8049 assembler embodied within a small microcomputer. The microcomputer and the experiment book accompanying it form the practical component of the course Microcomputers for Managers.

Layout of the Microcomputer

The microcomputer has a number of peripheral devices: lights to display the temperature when the micro is used as a digital thermometer; a temperature sensor, and two sets of red, yellow and green lights which simulate traffic lights. The microprocessor is in the centre of the machine. At the bottom are two sets of keys: control keys on the left, and data keys on the right. There are three red lights and one green light near the control keys, which are mode lights. Finally, above the two keypads are two sets of four display lights which display letters and numbers. These display what has been keyed into the microcomputer. The physical layout of this microcomputer is shown in figure 4.6 below.

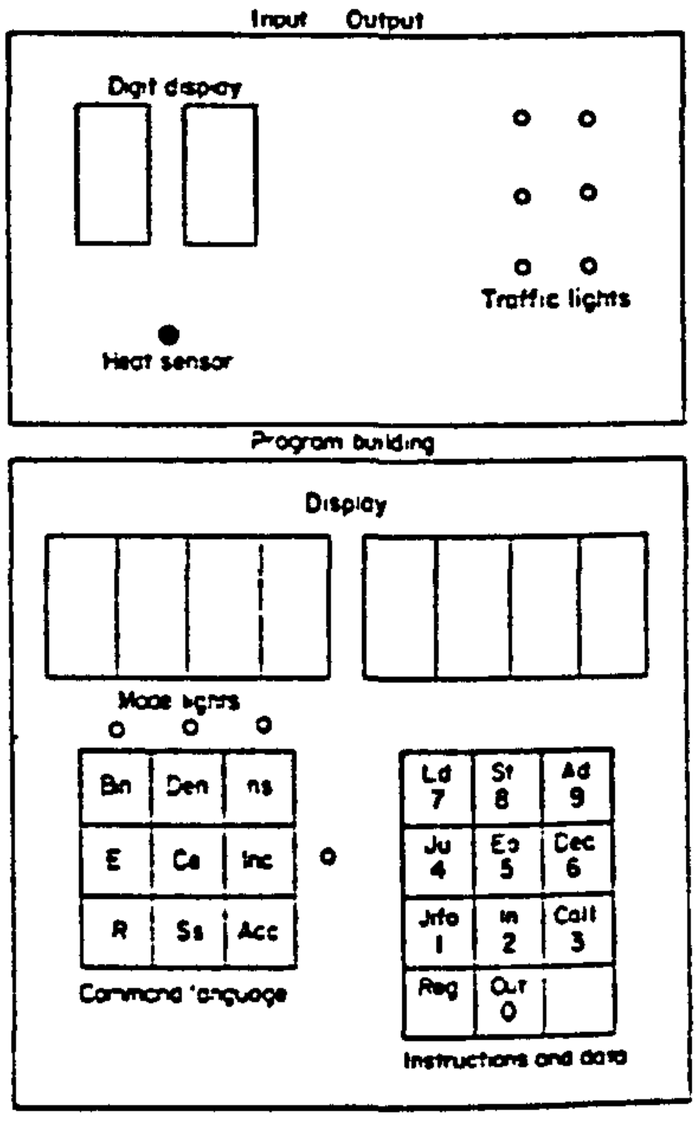


Figure 4.6: The PT501 microcomputer (from du Boulay, O'Shea and Monk, 1981)

The instruction set consists of 12 main instructions (counting LOAD and LOAD FROM REGISTER as one instruction etc) which are given in appendix 4.2. Three typical instructions are given in table 4.2 below:

STORE IN REGISTER	Stry	Copies contents of accumulator into register with address ry, replacing previous contents of register with address ry.
ADD	Ad xxx	Adds the number xxx to the contents of the accumulator, and leaves result in accumulator
JUMP IF ZERO	JIFO	Tests the contents of accumulator. If zero, puts xxx into the program counter, otherwise the program counter is increased normally

Table 4.2 Examples of PT501 instructions

4.12 PT501'S CONCEPTUAL MODEL: SIMPLICITY, VISIBILITY AND CONSISTENCY

Simplicity

The microcomputer has a small instruction set which is suited to the job in hand (to learn about the workings of microcomputers) and which the learner can use to do interesting things such as simulating a traffic light sequence, so in this sense it has functional simplicity. Functionally different keys are separated on the keyboard, so the instruction set, or keys for programming are clearly separated from the operational keys for editing, changing mode and dealing with programs (the command language). In the experiment book, the instruction keys are described as data keys, whilst the editing keys are described as control keys. The keys on the right hand section of the keyboard have to "double up"; each key includes both an instruction and a digit. In denary or binary mode the digit is selected, and in instruction mode the instruction is selected. Although this enables more to be fitted on to the keyboard it does make it more conceptually complex, as two of the

instructions are not included in the instruction set on the keyboard, and the user has to remember that these particular instructions have to be coded. These instructions are the RETURN instruction and the EXCHANGE WITH REGISTER both of which have to be entered in as denary numbers. The address pointer is also used as the program counter, and this violates the notion of simplicity, in requiring the student to take on two different perspectives:

"When our microcomputer executes an instruction it makes use of a special location called the program counter. This holds an eight bit address. First, the microcomputer reads the program counter's contents and interprets them as an address. Then it goes to the location in program memory with that address and reads the contents of that location. It then interprets those contents as the binary code for an instruction and executes the instruction. Finally it increases the contents of the program counter. It does this last step because lists of instructions (programs) are usually stored in locations with consecutive addresses. If the program counter's contents are increased each time an instruction is executed, the program counter will contain the address of the location which holds the next instruction in the list, which is obviously useful.
On your microcomputer system the program counter happens to be the address pointer. As you have seen you use the address pointer to select a location in the program memory and put in an instruction. When the microcomputer is executing an instruction it uses the address pointer as the program counter. This means that just before you get the microcomputer to execute an instruction you will need to make sure that the address pointer's value is the one the microcomputer will need in order to find and then execute an instruction....."

Visibility

There are three main ways in which PT501 can make its workings visible:

a) by making it clear which mode the machine is in

This is necessary in order to know what effect certain actions will have: PT501's three modes are binary, denary or mnemonic, and the mode lights indicate which mode the assembler is currently operating in, and so this shows one element of the state of the machine.

b) by making the contents of registers as clear as possible

This can be done, for example, by making it easy to inspect the accumulator, or a particular address and its contents. In the PT501 microcomputer the effects of

instructions can be seen: pressing the instruction/arithmetic key has an immediate effect. In denary mode, both the address and contents and contents of a location are displayed, but this is not the case for binary mode. The code of subroutines can be examined (although this is weak visibility) and programs can be single stepped.

c) showing the operation needed to change from state to state and its effect

The teaching materials introduce the notion of "state", and the novice is invited to navigate around a state transition network. The simplicity/visibility/consistency distinction breaks down here somewhat, in fact: the idea of the state transition network is to increase visibility by showing the machine moving from state to state, and so simplicity is required to make the state transition network comprehensible, and consistency is also required so that the correct keypress sequence can be predicted and remembered. Because the state transition network involves all three criteria, it will be discussed separately in the next section.

Consistency

Two features which have been discussed under simplicity and visibility also have implications for consistency. The fact that two of the instructions are not included in the instruction set on the keyboard is inconsistent (as well as making the keyboard more complex). The user cannot infer how she will use the two instructions which are not included on the keyboard, from her use of the other instructions, and this creates an extra cognitive load. Secondly, in denary mode, both the address and contents of a location are shown on the display, but in binary only the address or the contents are shown.

The state transition network

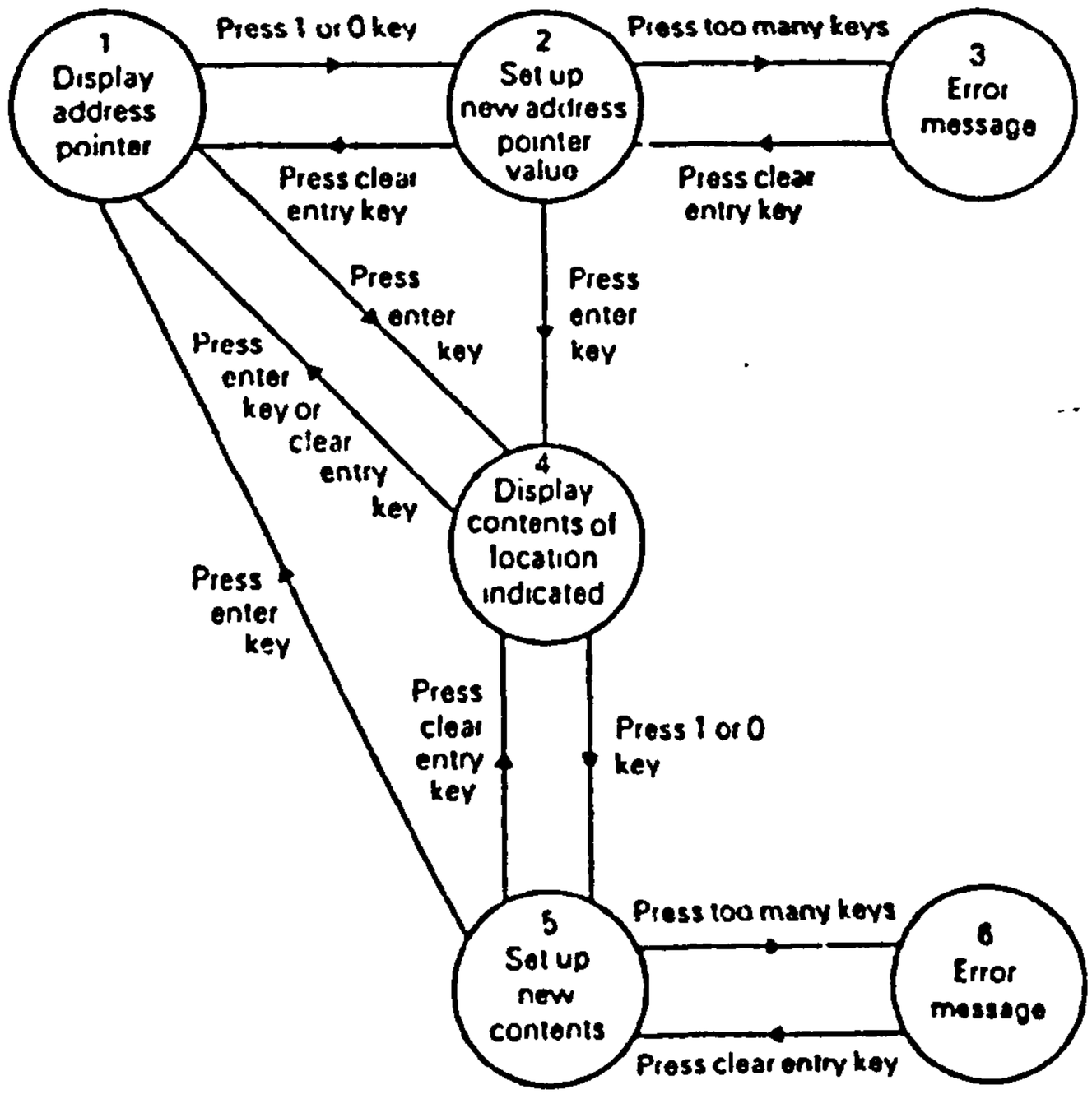


Figure . The process of selecting a location and changing its contents in the binary mode. The address mode light is on in the states labelled 1 to 3.

Figure 4.7 State transition network, from PT501 experiment book

The problems in navigating around this network are evident when the process (or operator) required for moving from one state to another is considered.

Consider the process of moving from state 1 through state 2 to state 4, and the implications for consistency:

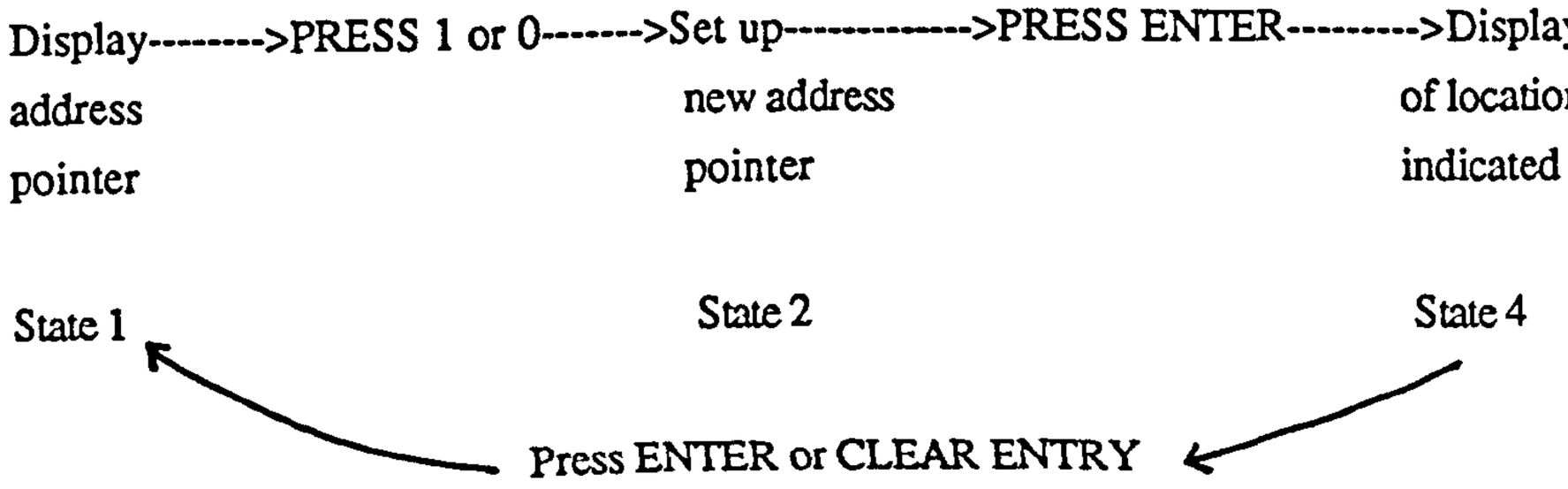


Figure 4.8: Moving from state 1 to state 4 via state 2

In this situation, a 1 or 0 is pressed to move from state 1 to 2, and then ENTER to move to state 4; however, to move back to state 1, ENTER is pressed again. It is not at all clear that ENTER will 'undo' the previous ENTER, and take the machine one state further back! Furthermore, ENTER and CLEAR ENTRY (which might be expected to have the opposite effects) are interchangeable. Another sequence also suffers from exactly the same problem. If the intention is to move from state 1 to 5, the picture looks like this:

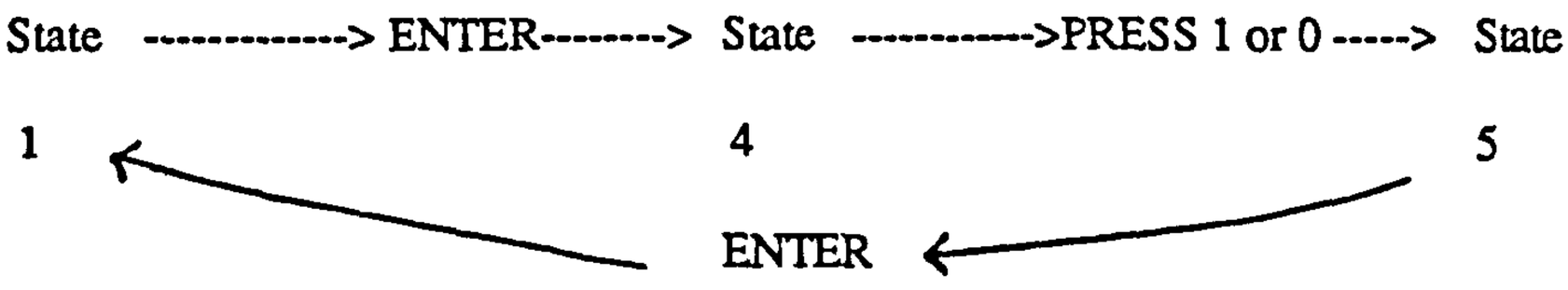


Figure 4.9: Moving from state 1 to state 5 via state 4

Here again the move from state 1 to 4 and 5 is by the operations enter, and 1 or 0, but to reverse the movement, exactly the same instruction is used: ENTER. This is the same as the first example given. The learner is going to have to study this network very carefully to discover the rules.

In terms of simplicity, although the idea is to help the learner see the states of the machine, and navigate between them, it is not particularly simple. One question is how may pathways are there to the same place? As can be seen, there are two pathways from state 1 to 4 and also two from state 1 to 5. And as has also been shown, there are several places where the same operator applies, but is used differently! Finally, different states, where different operators are needed, look identical. For example, suppose there is an error message, and the user has forgotten the context. She might reasonably assume that if she presses "CLEAR ENTRY" twice, this will take her back to the start state. This will work from state

3. If she is in state 6, she needs to press CLEAR ENTRY and then enter for the most direct route back (via state 5) or else CLEAR ENTRY three times.

Given these problems, visibility is clearly lacking, if the state transition network is not simple enough, and the operators are not consistent enough for ease of use.
In making other moves, the process is more predictable, as in the next figure:

State ------->Press 1 or 0 ------>State ----------> Too many keys---------->State

1 <------------Clear entry <------ 2 <-------- Clear entry <---------- 3

Figure 4.10: Moving from state 1 to 3 via state 2

Here CLEAR ENTRY is seen as reversing the effect of either pressing a 1 or 0 or pressing too many keys (remember that state 3 is an error state). The "WHICH" rating of PT501 is:

Simplicity	**
Visibility	***
Consistency	*

4.13 PT501'S CONCEPTUAL MODEL: STATE, PROCEDURE AND FUNCTION

The PT501 machine operates in three modes: denary, binary or mnemonic. The particular mode in which the machine is currently in is shown by the appropriate mode light. Thus the mode is explicit (denary, binary, mnemonic) and also the level of operation: dealing with the program (editing) as opposed to the instructions that comprise it. It shows the address or contents of a program location or the contents of the program counter. The contents of the accumulator can also be viewed. All these highlight the state of the machine.

The teaching materials introduce the notion of "state" and supply a state transition network for the novice to navigate around. Given that this includes the operators which act to change the state, this is a procedural representation. (See fig 4.7 above). A procedural description is also given by stepping through procedures - where the starting state is known. The intention is that giving both a state description and a procedural description together helps the learner to get a complete picture of what is happening. However, there isn't much of a functional description, at any level, other than explaining what instructions do, and even this is not apparent until later in the experiment book. It may be that a functional model is not appropriate: the learner is not invited to build up programs (as with DESMOND) but is given detailed step by step instructions to take her through the experiment. Programs do not appear until experiment 18. Much further on in the experiment book, there are more functional models, for example, section 6.3, a multiplication program, describes a counter (though it doesn't summarise the steps) in a way that emphasises the <u>function</u> of the instructions:

"It would be useful to have a list of instructions that would multiply any two given numbers together if those numbers were held in two registers. This can be done by using a little group of

instructions which the microcomputer can execute repeatedly. For instance, to multiply three by six the microcomputer could execute six times the same group of instructions which included an instruction to add three. Suppose that the register with address r1 holds one of the numbers to be multiplied -say, six. Then each time the little group of instructions was executed the contents of the register with address r1 could be decreased by one. When the contents of the register were zero then the microcomputer could stop executing the group of instructions: it would have executed them the six times it needed to do so. The register with address r1 is then acting as a counter which counts down each time the little group of instructions is executed until it reaches zero. The little group of instructions plus the instructions needed to make the count down possible are called a loop of instructions. ....."

This is a good example of a functional view, which the learner can relate to a plan. It clearly addresses the question "How can I multiply two numbers together?" However, this kind of explanation and description is unusual is PT501. The learner is taken through the detailed steps of the experiment without any plans being made explicit. Another part of the manual explains an instruction in the following way:

"You have just discovered that the contents of the stack pointer are increased by one during the execution of a CALL instruction and decreased by one during the execution of a RETURN instruction....."

This is also a functional view.

4.14 SUMMARY

The microcomputer has a small instruction set which is suited to the job. Functionally different keys are separated: i.e. the instruction set versus the editing keys, and the effect of the keypresses is not always consistent. However, the whole instruction set is not included on the keyboard. The mode lights which indicate which mode the machine help visibility and in denary mode, both the address and contents are displayed. The contents of the accumulator can be displayed, but they are not shown automatically. One aspect of visibility is the state transition network, but this is not particularly simple, and the learner will have to study it carefully to discover what they need to do to move from state to state. As far as the different types of conceptual model are concerned, the emphasis is on both state and procedural descriptions (as shown in the state transition networks) with little emphasis on the functional description.

4.15 DESMOND

DESMOND is the name given to the microcomputer used as part of two Open University study packs: Inside Microcomputers and Learning About Microelectronics (Open University, 1985). In this study, DESMOND was only used in its computer mode. DESMOND is in fact a computer simulating a simpler computer! Its central component is a Motorola 6805 single-chip microcomputer. It has additional memory and circuits which provide interfaces for some of the inputs and outputs. Like the PT501 microcomputer, the simulated DESMOND processor contains three registers, an Accumulator, a flag register and a program counter.

DESMOND has 16 instructions, - a sub-set of the 6805 instruction set. The 16

DESMOND instructions are given in appendix 4.3. The three instructions given below are typical:

LDI	LDI xxx	Load Accumulator with xxx
ADD	ADD xxx	Add to Acc from mem location xxx
JZ	JZ xxx	Jump to mem location xxx if zero flag set to 1

Table 4.3 : Examples of DESMOND instructions

DESMOND is described in various ways in 4 different documents. These documents are the study guide which directs the readings and activities of the student; the practical book which contains all the DESMOND practical activities and is the main document used in this study; the reference card which accompanies DESMOND and summarises all the instructions and gives the layout, memory map etc., and the "Glossy Book", so called because it was designed to look like an attractive coffee table book. In this study, students only had access to the DESMOND practical book. The book uses metaphors and makes links with everyday life. In the study guide it is described as follows:

"To understand the action inside a computer, it helps if you have some visual images and metaphors to link it to your everyday experience. This book attempts to give you that by using a highly-illustrated, magazine-type format."

The analogy given for the computer as information-processing system is the office, where the manager's office corresponds to the computer's processor, information storage is in the library or the computer's memory and so on. Parallel transmission is illustrated by four ranks of soldiers crossing four bridges whereas in serial transmission the ranks have to break and cross the bridge in single file. Finally there is the DESMOND microcomputer itself, which is shown in fig 4.11.

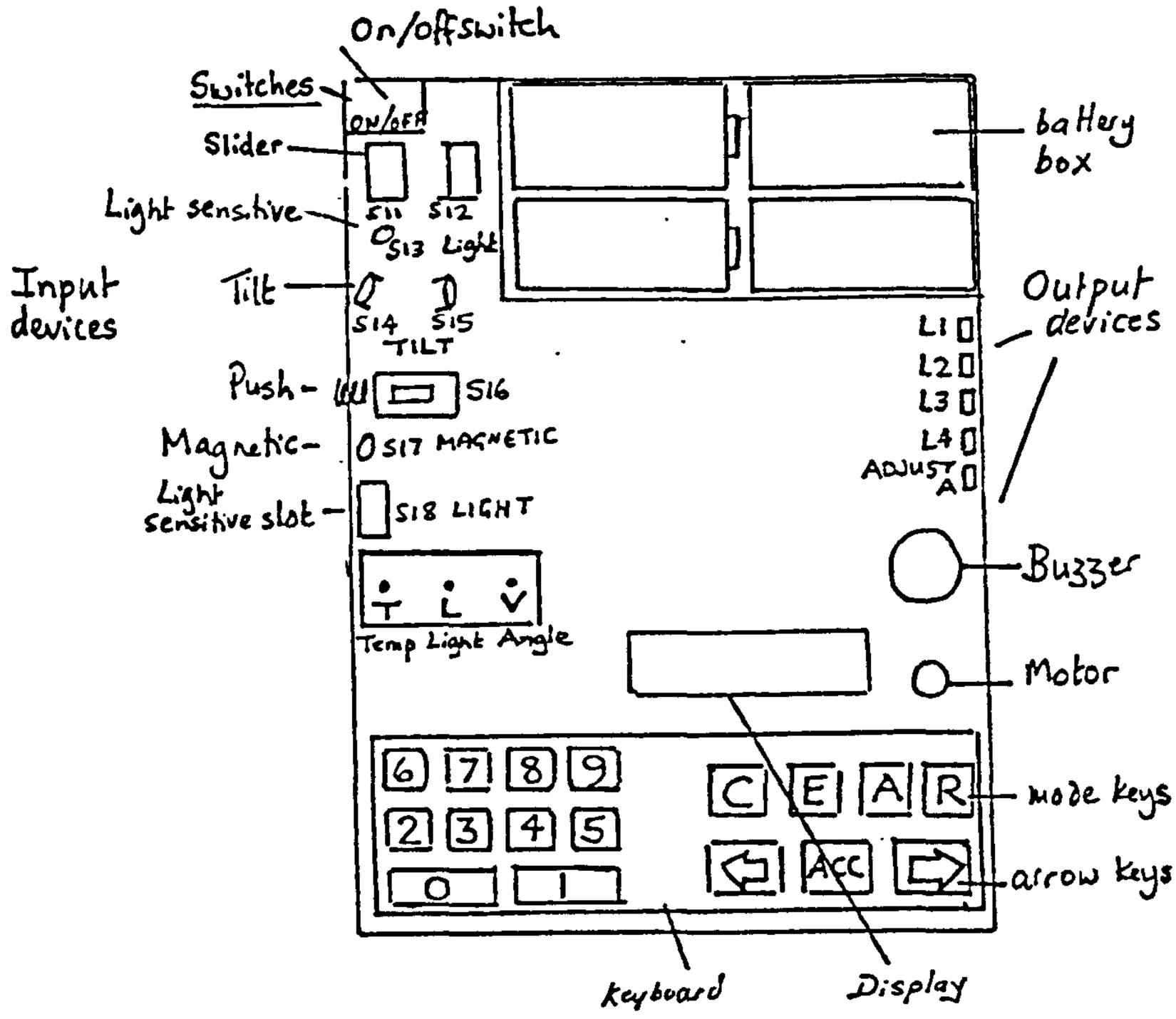


Figure 4.11: The DESMOND microcomputer

The battery box is on the top right. On the top left are various switches: the on/off, slider, light sensitive, tilt, push, magnetic and the light sensitive slot. Below these are the sensors: the temperature sensor, an angle sensor, and a light level sensor. On the right below the battery box are five lamps (red yellow green red and adjustable). Below this is the buzzer and below this the motor. As with PT501, all the circuitry and chips are visible. Unlike PT501, the instruction set is not on the keyboard. There are two sections to the keypad: the left hand side contains digits from 0 to 9, and the right has two rows of keys, of which the top four are described as mode keys:

"When turned on, DESMOND will always be in one of four modes. To change modes, press the appropriate key"

They are: C to choose the microcomputer rather than the microelectronics course and to go into monitor mode where DESMOND shows the value stored in hex., denary and binary; E to erase a program; A for assembly mode where programs can be entered in mnemonic form and R for running a program. The function and

effect of each of the 4 keys described above is then given. It is clear that C, A and R change the mode, but describing the erase key as a mode key seems rather odd.

4.16 DESMOND'S CONCEPTUAL MODEL: SIMPLICITY, VISIBILITY AND CONSISTENCY

Simplicity

Like PT501, DESMOND has a small instruction set and is well suited to its task. The layout of the board is clear, and so it has a high claim to simplicity.

Monitor mode

The display is divided into three parts. In monitor mode the first part of the display gives the address of the memory location, the second part gives the content in denary and the third part gives the display in binary. Arrow keys, (the next row down from the mode keys) are used to move backwards and forwards through memory locations (See fig 4.12).

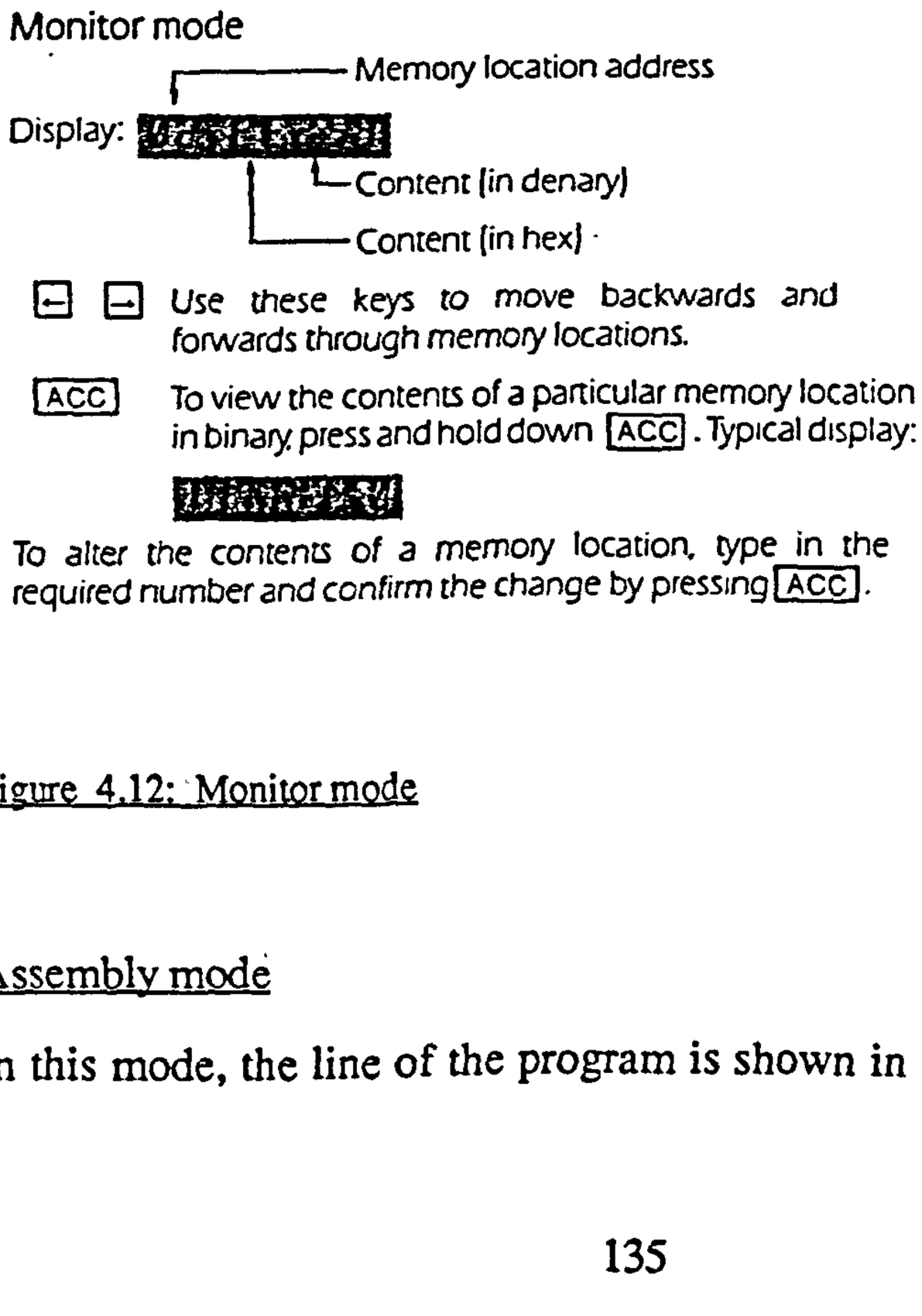


Figure 4.12: Monitor mode

Assembly mode

In this mode, the line of the program is shown in the first part of the display, an

instruction in the second, and 3 digits in the third. All 3 segments of the display can be altered in turn. Which segment is currently active is indicated by it flashing. The arrows are used to move through the different program lines or instruction set but not to change the final digits which form the third segment of the display - as this would be too cumbersome. So the arrows are used to move through either a) lines of program, if the address is flashing or b) the instruction set, if the instruction is flashing. The ACCept key is used to confirm the section that is flashing (address, operation or final number). The display is shown as a wheel, with the display being the segment that is currently in view .

Run mode

Here the arrows are used to move between GO and single stepping (SS), and ACCept is used to confirm this choice. However, in single stepping mode, they have a different function, where -> executes the line being displayed and <- shows what would have been on the display in normal fast run mode. ACCept shows the current contents of the Accumulator and the state of the flags. The 3 segments of the display show the zero flag, the lower flag and the contents of the accumulator.

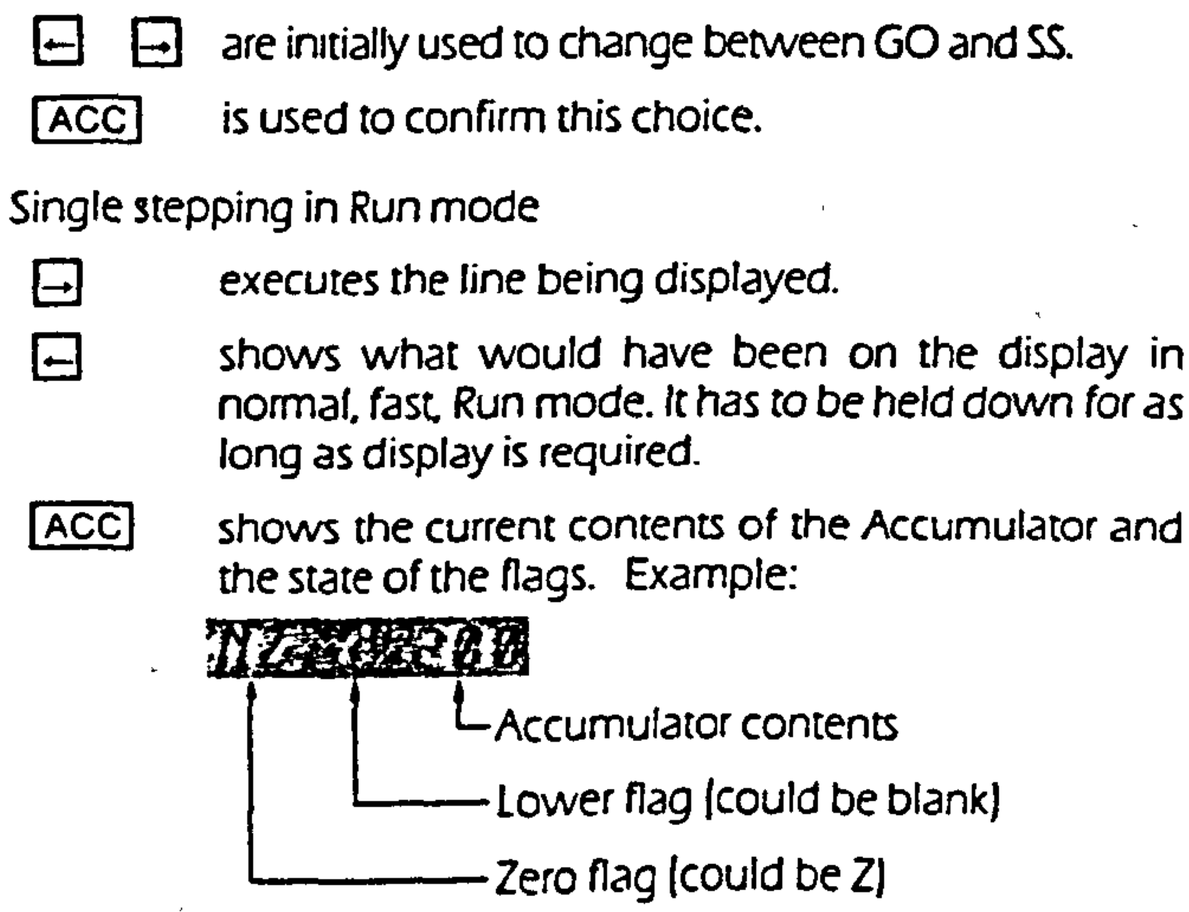


Figure 4.13: Run mode

Visibility

There are three ways in which DESMOND can make its workings visible:

a) by making it clear which mode one is in

By mode here is meant monitor, assembler, run or erase. Unlike PT501 there is no need for mode keys, as it is clear whether one is in monitor, assembly or run mode. Monitor always displays the address and its contents in two forms, hex. and denary. Assembler always displays the program line, the instruction, and when the instruction isn't NOP, a number (the code). Run mode always initially displays either Go or SS, then when running in normal mode, three noughts or whatever has been programmed to be on the display; or in single step mode, the address and contents as it would appear in monitor mode, or what would have been on the display in normal running mode. As is evident from the number of words needed to describe the various options in run mode, it is this mode that has the most risk of confusion, but this is a cognitive load problem rather than a problem of signalling the mode. Earlier, the 'error' mode was signalled as a problem, in that it is dissimilar in type to the other modes. Strictly speaking it is a mode, and re-initialises User Memory.

b) by making the contents of registers as clear as possible

Examples of making the contents of registers visible might be making it easy to inspect the flag(s), the accumulator, or an address and its contents. DESMOND is very good here and the display gives as much information as one could reasonably expect. In monitor mode, the display shows the address of the location and its contents; in assembly we see the program line, and the two parts of the instruction, the operation and the number. The normal run mode does not show contents, but single stepping allows us to see both the locations and their contents, and, importantly, what's in the accumulator. The exercise on using the single step mode, and then answering questions about which instruction has just been executed and the current contents of the accumulator etc. indicated that despite these facilities, students have problems tracing a program's execution in a detailed and accurate way. There are two likely reasons. One is that there are far more options

in run mode, and as was seen earlier, the effect of the arrow keys and the ACCept key is not consistent with their more usual use. Secondly, the notion that what is currently on the display has not yet been executed appears to be problematic, especially where there are jumps, so that the cognitive load of remembering where one is, is also high, in other words although the display shows the location jumped to, which is the line about to be executed, this is not put in the context of the rest of the program. To get a total picture, therefore, it would be necessary to do this at the same time as looking down a list of the program.

c) by showing the operation needed to change from state to state and its effect

One important kind of state in DESMOND is the mode which the user is currently in. How to change mode is summarised on the fact card and also which keypresses to make and their effects once in a particular mode (see fig 4.14 below).

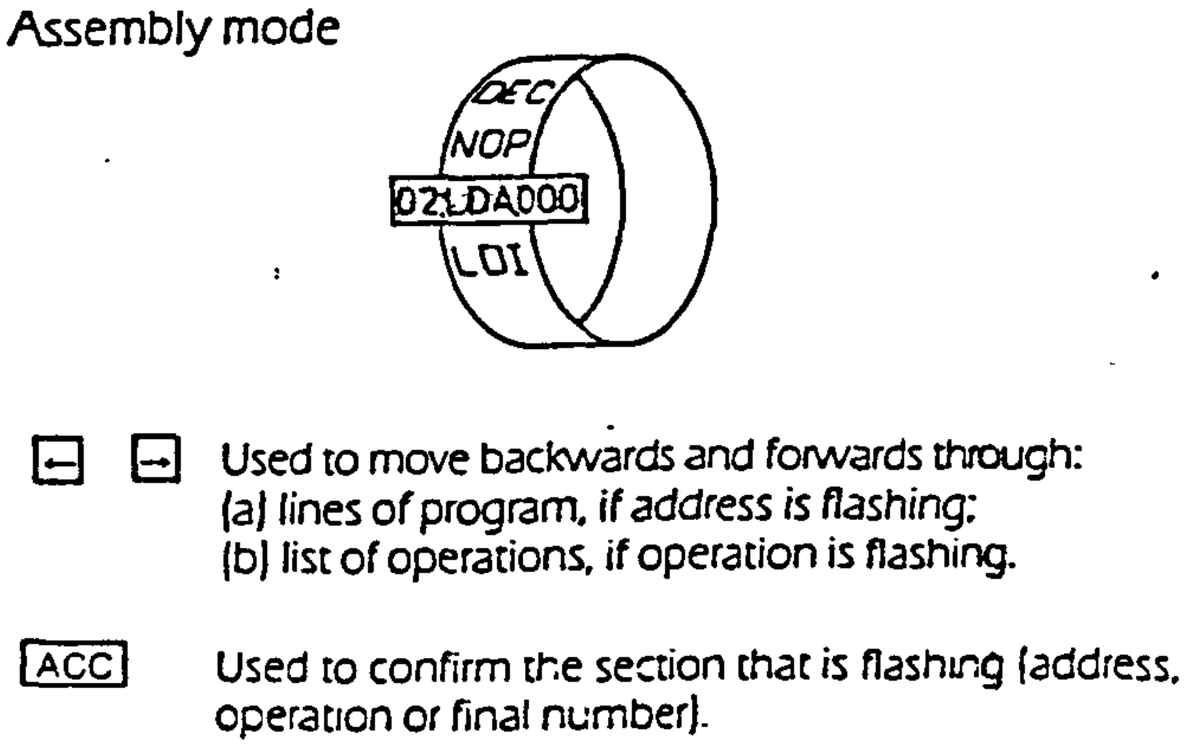


Figure 4.14: How to move around in assembly mode

Keypresses are given for most of the experiments in the practical book, except for the final experiments where it is assumed that this level of guidance is not needed. The instructions are given in a three column format where the first column gives the keypress, the second shows what will appear in the screen and the third gives the commentary. (See fig 4.15 below)

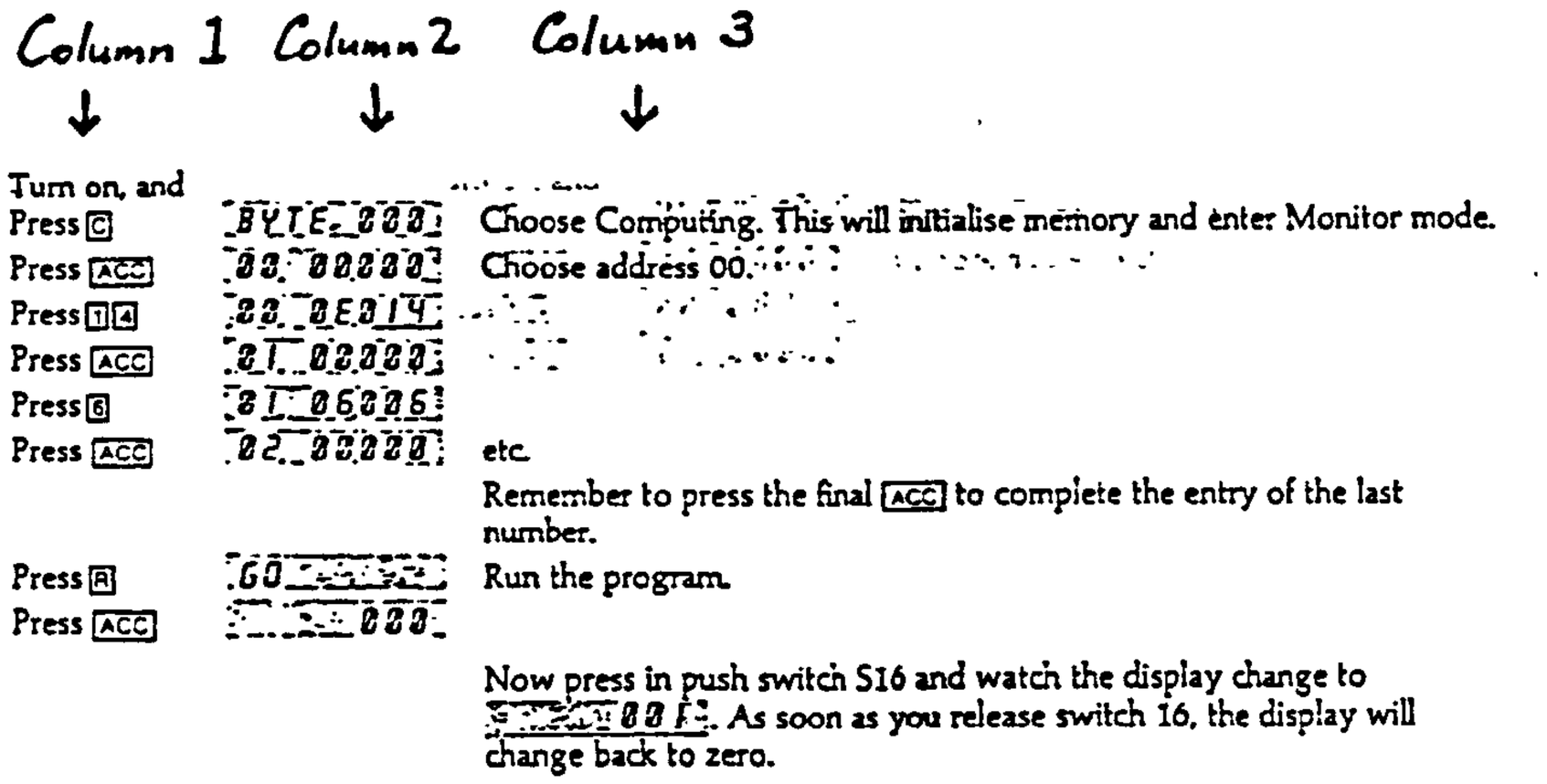


Figure 4.15: Three column format

There is a lot of information on changing state; but unlike PT501, it is not summarised in the form of a state transition network.

Consistency

DESMOND violates the consistency principle: - presumably to make it possible to provide a number of facilities with a limited number of keys and instructions. For example the arrows are used in monitor mode to move backwards and forwards through memory locations: similarly in assembly they are used to move through the instruction set or the lines of program. In run mode however the arrows are used initially to change between GO and single step (SS); then once GO has been confirmed, (using ACCept) the right arrow executes the current line, whilst the left arrow shows what would have been on the display in normal run mode. There are two violations here. First of all the arrows are used for a different purpose, i.e. changing between two modes of running a program, or between execution and showing what would have been on the display as opposed to moving forwards or backwards through different 'locations' of some kind. Secondly in run mode, the effect of pressing an arrow differs according to the stage of the operation, i.e. initially it is a toggle between GO and SS, but after pressing ACCept, -> executes

initially it is a toggle between GO and SS, but after pressing ACCept, -> executes the line and <- shows what would have been on the display.

One question is whether the consistency criterion should include control or editing keys such as the arrows. One argument for it being important, even in control applications rather than programming, is that if such use is consistent, one application can be used by analogy with another. This works well with the Macintosh, and is not confined to programming activities (although it can be argued that the distinction between programming and control is more blurred on the Mac.), and transfers between applications. For example, a user can assume with an unknown piece of software that she can take the mouse to one of the headings at the top, many of which stay constant across applications, click to obtain the menu, and drag the mouse down to where she wants and release to activate the appropriate action. In other words using a new application is done by analogy with the old, known application. With DESMOND however, applying the fruit machine type wheel analogy to the use of the arrows, breaks down in run mode. Toggling between GO and SS could be presented as a limited two state wheel, but once run mode has been confirmed, the arrow's effects cannot be presented in this way.

Another breakdown of consistency is in assembly mode, where for two segments the arrows are used to move through the available choices, but in the third segment, the number is simply typed in.

Three violations of consistency have been identified, all of which are to do with using the arrow keys:

1 In assembly mode the third segment of the display is altered by direct typing, not by moving through different states

The consequences of assuming that the arrow keys are used to move through the third segment are not far reaching. Pressing one of the arrow keys when the last segment is flashing results in an error buzz. Once some time has been spent moving through the instruction set (which is 16, so any instruction takes a maximum of 7 presses if you get the direction right) it is clear that using this mode for accessing a large number set would require too many key-presses.

2 The arrow keys are used differently in run mode generally

The first different kind of use in run mode is not problematic: the arrows can be seen as toggles through two states - (although in fact there is a slight added complication: nowhere does it say that **both** arrows toggle between GO and SS). Learning or remembering the second use of the arrows, (to execute or view what would have been on the display) will require more effort as it breaks with the previous analogy.

3 The specific effect of the arrow keys in run mode depends on the starting state

There are two possibilities depending on the stage of the operation. Whether the different use of the arrows at different stages is confusing will depend on whether the current state is clear. When GO has been accepted, and the program is running, the display will either show three noughts, or the display that has been set up in the program. In single step mode, however, when ACC is pressed, it shows the first address and its contents (the instruction), and executing this line by pressing the right arrow, results in the next line being shown. The different states, then, from which the arrows might be used are quite clearly delineated.

These violations of consistency here do not appear to have far reaching consequences. Using the arrows for moving through various states or instructions, obviates the need for an instruction set on the keyboard (as in PT501), or, even harder, having a large enough keyset to have the letters available for typing in the assembler mnemonics. Incomplete consistency, therefore, is the price for a conceptually clean keyboard. The other price, of course is the larger number of key presses than if each instruction had a corresponding key. Many such tradeoffs will be needed where there is limited size and power. The important issue is to consider the consequences of violating such principles, and if there is any possibility of confusion, how easy is recovery?

The arrow keys have been discussed at length. The other key on the same row is the ACCept key. This is generally used to confirm, or accept what has been typed or is on the display. However, in one instance it is used differently. This is in single step mode where pressing ACC shows the contents of the Accumulator and flag for as long as the key is pressed. Whilst this is a useful facility, it is not consistent with its other use, and it also increases any possible confusion between the use of ACC as ACCept, and as something to do with the ACCumulator. DESMOND's "WHICH" rating is:

Simplicity	****
Visibility	***
Consistency	**

4.17 DESMOND'S CONCEPTUAL MODEL: STATE, PROCEDURE AND FUNCTION

Three ways were mentioned for making the DESMOND microcomputer's workings visible, which are:

1 making the mode clear

2 showing the contents of registers, and

3 showing immediately any changes of state.

They help to make the state of the machine visible and give a state description. In address mode one can view the address, operation and operand, so this gives a procedural description, as it gives the user a view of changes to the machine's state.

Instructions on DESMOND

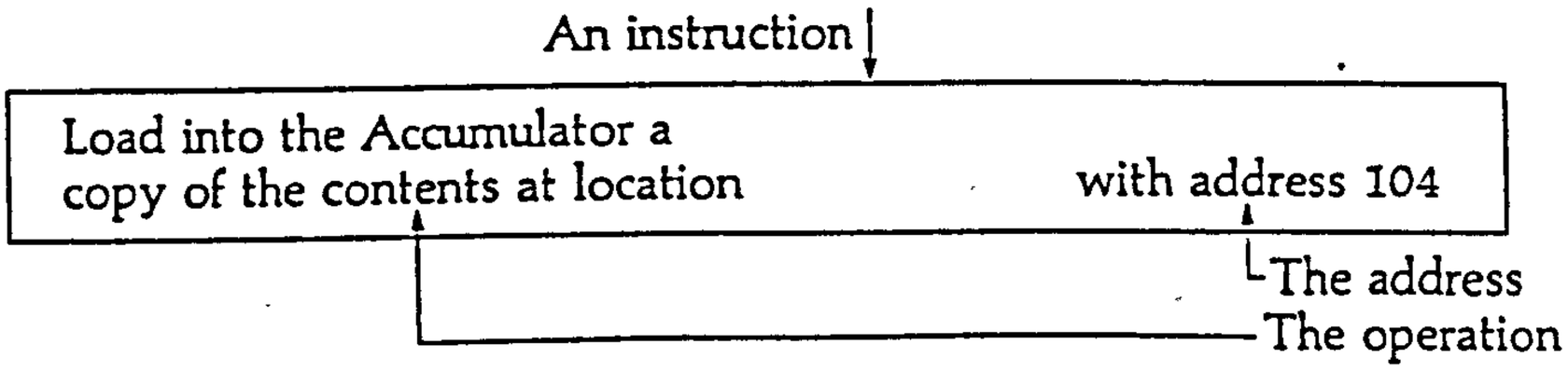
Every instruction on DESMOND is coded as two numbers, which means that each instruction uses two adjacent memory locations. Here are the codes for the program:

1 Load into the Accumulator a copy of the contents at location 104 : 015 104

2 Go to the DESMOND binary display routine at address 205 : 007 205

3 Jump back to the start and repeat the program : 010 000

Each of the three instructions given above divides into two parts: an operation and an address. Consider the first instruction:



This is coded into two numbers. The operation 'Load into the Accumulator a copy of the contents at a specified location' has been given (by the designers of DESMOND) the code: 15. This means that if DESMOND encounters the code 15 when it is expecting an operation it will know which operation to perform. (Other machines might code the same operation in quite a different way.) The second of the two codes for this instruction is simply the address of the memory location from which a copy of the contents will be taken.

Figure 4.16: A procedural view

The DESMOND practical book contains analogies such as a row of houses for memory locations (state description). The programs are given key press by key press and the resulting state is shown, so these are procedural models. As with PT501 there is little in the way of a functional description.

The emphasis in the description and diagram given for instructions on DESMOND (see fig 4.16 above) is mainly procedural. There is no start state given, but there is enough information to work with whatever the start state is. In chapter 9, DESMOND is analysed in terms of a number of plans, and it is argued that in order to learn DESMOND it is necessary to master these plans, which can be thought of as high level functional models. Yet DESMOND is not presented in a way which emphasises such a functional description.

4.18 SUMMARY

DESMOND has a small instruction set and is suited to its task, and the layout of the board is clear and accessible. Arrow keys are used to move through the registers and the instruction set, thus obviating the need to have instruction keys on the board. It is clear which mode one is in from the context, and DESMOND's display gives quite a lot of information about what's going on inside (e.g. if the contents of the accumulator have changed). Remembering the operations needed to change state is helped by having the arrow keys used both in monitor mode and assembly mode. Use of the arrow keys in assembly mode, however, does necessitate a larger number of key presses than if the instruction set were on the keyboard, and there are some violations of consistency in using the arrow keys, but these do not appear to have far reaching consequences. The conceptual model gives more emphasis to procedural and state descriptions than to functional descriptions.

4.19 A COMPARISON OF PT501 AND DESMOND

In the same way that Logo and SOLO are doing a similar job as high level languages, both PT501 and DESMOND aim to introduce novices to the workings of microcomputers and were both designed as teaching devices. PT501 was produced before DESMOND however, and the experience of producing PT501, and feedback on its use, has led to DESMOND being the better product.

Both have similar sized boards, laid out so as to make the workings accessible and conceptually simple. PT501's set of instruction keys, (which double as data keys) are physically separated from the editing keys. Unfortunately the whole instruction set isn't included. The DESMOND keyboard is rather neater and is simpler. As the arrow keys are used to move through registers and instructions, no instruction keys are needed and so the data keys are separate and there are just four mode keys and an ACCept key in addition to the two arrow keys. However, more keypresses are needed to input programs. As the modes are clearly different no mode lights are needed. Both machines have the input and output devices separated. DESMOND has a larger set of both and therefore there are a greater variety and set of exercises for the learner to carry out. Both machines have reasonable visibility but DESMOND's three segment display always shows addresses and their contents, whereas PT501 only has this facility in one mode, and the state transition network provided in the PT501 experiment book isn't particularly clear or simple to use.

Although DESMOND is inconsistent at times, this does not seem to be too damaging. Altogether, DESMOND's model is simpler and more visible. Both DESMOND and PT501 emphasise procedural and state descriptions as opposed to functional descriptions.

4.20 SUMMARY AND CONCLUSIONS

This chapter has described the four languages to help the reader to follow the examples that will appear in chapters 6 - 9. The conceptual model of each has been evaluated according to the criteria suggested by du Boulay, O'Shea and Monk (op. cit), and additionally, according to the different ways of describing the conceptual model which was discussed in chapter 3. The languages were then considered in pairs: SOLO and Open Logo, and PT501 and DESMOND. Although each was designed for teaching novices, SOLO and DESMOND were considered to be more successful than others and to have 'better' conceptual models. This is confirmed, informally, by the "WHICH" rating which is as follows:

	SOLO	PT501	Logo	DESMOND
Simplicity	****	**	**	****
Visibility	**	***	***	***
Consistency	***	*	**	**

It is not possible to "rate" the different ways of describing the conceptual model in the same way, nor to add up the ratings. However, none of them stress a functional view and so it might be predicted that novices will find constructing plans difficult, especially for the low level languages PT501 and DESMOND which give least emphasis to a functional view.

Chapter 5

METHODOLOGY

Contents **Pages**

5.1 INTRODUCTION

The main concern in this thesis is to investigate how novices learn to use languages taught using conceptual models, and this chapter discusses the methods that are appropriate for this task. The rationale for the approach taken is given, and evidence is presented for the soundness of the techniques used. A more detailed account of the methods used is given in the appropriate chapters.

Different methods and types of data were suitable at different stages and adopted accordingly. The different sources of data included repertory grids, different programming exercises and problems, interviews, protocols and comment sheets. Ways of collecting and analysing these sources varied accordingly. This section discusses the rationale for using one of the methods of data collection which was used to collect much of the SOLO and PT501 data, and its validity. Details of methods of collecting data for particular studies will be found in the appropriate chapters.

5.2 RATIONALE

One aim of this research is to improve instructional materials for teaching at a distance, and in order to do this, it is necessary to gain an understanding of the cognitive processes involved in learning to program, which can then be utilised in the teaching process.

The theoretical framework for this study has already been described as a phenomenological approach which is shared by other educational disciplines, for example science and mathematics education (e.g. Gilbert and Watts, 1984; Novak and Gowin 1983). Although cognitive science has evolved from a rather different

context, it often involves a methodology which in many ways is similar to the methods used in the approaches mentioned above (e.g. Newell, 1977 (op. cit.)). This is not surprising as in attempting to model learning behaviour it is clearly necessary to collect detailed data about how learners behave. A standard artificial intelligence technique involves analysing protocols and using this data to construct production rule models which can then be used to predict the kinds of errors students make. Such models are valuable in their own right for examining learning and they can also be embedded within tutoring systems (see for example O'Shea's (1979) Self-Improving Quadratic Tutor).

There are two main methods for investigating learners' cognitive processes. The first is to design experiments which test certain hypotheses. The second is more akin to a field study and takes the form of observing an existing situation, which allows hypotheses to be formed, rather than to be tested. Whilst experimentation can be very valuable in revealing the importance or existence of certain effects, it was not appropriate here for the following reasons:

1 The aim of the research was to investigate the problems and difficulties which occur when novices are learning a programming language with the aid of a conceptual model. It is not possible to predict in advance what the problems are, and so the method used must allow the researcher to see what emerges from the data.

2 There was no intention to investigate a particular theory or model - for which experimentation would have been more appropriate. Although the investigation does make theoretical assumptions (see section 1), no 'theory of learning programming' was being proposed.

The intention was to collect data in order to see what difficulties arose, and from

this, to observe any patterns, so as to categorise the data and explain it. In an article entitled "The structural paradigm in protocol analysis" (Easley 1974) where Easley argues for protocol analysis of clinical interviews as a method for identifying and investigating cognitive structures, he states:

"It would appear reasonable, with regard to these views (referring to Kuhn) that some insight into structures underlying observed phenomena is needed before we can reasonably hope to define measureable quantitative techniques appropriate for further illuminating and systematising our psychological observations."

Such an approach is consonant both with artificial intelligence techniques as described earlier, and with grounded theory (Glaser and Strauss, 1968), the notion of which is that the theory is 'grounded' in the data:

"The basic theme in our book is the discovery of theory from data systematically obtained from social research."

Although the research in question here is not social research, the same principles apply: of looking for patterns in the data and generating categories to start explaining the data, i.e. discovering theory.

The approach taken was to use a number of different methodologies, from within the frameworks described above. There is a role both for in-depth protocol analysis and experimental paradigms in this area: an in-depth study can illuminate what is going on in the situation, and can lead to models of the learner's behaviour. Once we have such models then variables can be manipulated in the more traditional experimental paradigm to uncover various influences. In this thesis, no experiments as such were carried out, although the study of LOGO and the DESMOND assembler could be classified as such in that the dependent variable was exposure to either DESMOND or LOGO. However, the later studies backed up the open-ended protocol analysis of the earlier work with interviews, error analyses and students' open ended commenting on the curriculum. The main

methodology used for a large proportion of the data was protocol collection and this is described in the following section.

5.3 PROTOCOLS

A general definition of a protocol is a detailed record of a subject's performance on a task, although a variety of different types of 'detailed records' are described as protocols. For example, some investigators record the direction of a subject's gaze (eye movements) while performing a task, some record details of the task performance e.g. key strokes in a calculator activity (behaviour stream) and some concentrate on recording what people say (verbal data). This last category in itself contains a variety of types of report. Experimenters are trying to probe verbally the subjects' internal states. One way is to instruct them to think or talk aloud. A procedure related to this is concurrent probing where the subjects are probed concurrently with their performance of a task for specific information e.g. "what are you thinking now?" Another type of verbalisation procedure is retrospective probing - asking the subject for information just after the completion of the task or after an experimental session with a number of different trials. This is called *interpretative probing.*

Validity of the data

Verbal reports such as these have often been dismissed as mere introspection (Nisbet and Wilson, 1977) or rather as fundamentally suspect as data: the argument being that introspection can be useful for discovering possible psychological processes but needs to be verified by some other sort of objective measurement (Lashley, 1923). However, it has been pointed out (e.g. Breuker, 1982) that such concerns apply equally well to other data such as response times which are considered to be more hard, objective data. In an extensive literature review,

Ericsson and Simon (1981) discuss the evidence for each of three concerns which are:

1 that the cognitive processes are affected by the instructions to verbalise;

2 that the information is not available to be verbalised, and

3 that information from different protocols from the same subject is not consistent.

An analysis of their findings is given by Jones and Scanlon, (1983) but their conclusions for each of the above questions are that:

1 instructions to verbalise and probe do not alter the course and structure of cognitive processes but instructions which require subjects to recode information in order to report it (i.e. produce reasons) may do so;

2 results from concept learning studies suggest that the subjects had more information at the time of their original experiment than they gave in their verbal reports. It seems, therefore, that the evidence from protocols is not complete, and

3 inconsistencies could arise in two main ways: subjects asked to access long term memory could recall information related to, but not identical to the information required, or some intermediate processes could fill out and generalise incomplete memories before responding.

Both of these factors would affect retrospective rather than concurrent protocols. From the literature they review they suggest that concurrent verbalisation, therefore, appears to be the most robust strategy.

Ericsson and Simon conclude with some general recommendations for using

protocol data which are:

1 Use more than one set of data where possible. It then becomes possible to check consistency of verbal reports with other data - the triangulation method.

2 Use concurrent verbalisation. Inconsistent retrospective reports can be produced as a result of students' use of inferential processes to fill out and generalise incomplete or missing memories.

3 Use free responses. Inconsistent reports can be produced as a result of probes that are too general to elicit the information actually sought.

4 Avoid specific probes (like "Did you use X as a subgoal?" or "Did you use any subgoals? If so which?") These create data.

Other writers also conclude that verbal reports are a valid form of data, for example Breuker (op. cit):

"To put it simply, one should not worry too much about validity of verbal data, particularly not because on one hand there is a lot of experience about the conditions under which one can expect reliable self-report, and because on the other hand, a posteriori indications of validity can be collected in the first place by analysing the protocols and constructing a theory that works -or not-."

Ericsson and Simon's recommendations which were given above were followed in carrying out the studies of PT501 and SOLO where protocols were the main source of data, and subjects completed other tasks such as programming construction tasks and prediction tasks which provided error data.

Protocols included both verbal data and keypresses wherever possible and concurrent verbalisation was used. In order to elicit free responses from subjects and to avoid specific probes (see 3 and 4) the experimenter left the room so that subjects were not tempted to ask for help and she wasn't tempted to give it or to use inappropriate probes or ask leading questions. Subjects were given a short problem-solving task before starting their work on SOLO or PT501 so that they could get used to thinking aloud,

Capturing the available data

A further issue is that of completeness: whether the particular method of data collection captures the available information. The author (and colleagues) conducted an investigation into several different technologies for recording protocol data of varying kinds as part of a study on the viability of producing production system models of various kinds of behaviour (O'Shea et al, 1985). One of the dimensions on which the recording technologies were evaluated was completeness. The recording technology of interest here is the Cyclops system used to collect data on subjects programming in SOLO and the PT501 assembler. (Audio-tapes were used to record interviews of subjects learning LOGO and DESMOND in chapters 9 and 10 - but these were not the primary sources of data).

The Open University Cyclops system enables voice and data to be recorded simultaneously: the data here is the subject's transactions with a computer. The voice and data are synchronised and can be played back in real time. The conclusions of our investigations were that although video is far more expensive than Cyclops it does not necessarily provide much more complete relevant information. Behaviours such as eye movements are missing, but these are not relevant to the issues pursued here, and the 'immediacy' of the video is lost. There is also the question of what is happening during the pauses; which the video gives some chance of finding out.

Two kinds of programming data were considered in the study undertaken by O'Shea et al: solving programming problems, and recording data on programming language constructs; and for both of these tasks, it was felt that the technology performed well in capturing the data.

5.4 OTHER METHODS

In line with Ericsson and Simon (op. cit)'s recommendations for using protocol data, additional data was collected where possible. In the first PT501 study subjects were sent questionnaires on which they were asked to explain concepts and also to group them, and SOLO students were interviewed at summer school.

Concept grouping

Chapter 2 discussed the importance of the organisation of programming knowledge, and how the organisation of such knowledge varies between experts and novices. Some indication of how a novice organises the concepts of a programming language, therefore, used in conjunction with other data, may yield some information about the level of their understanding. In the first PT501 study, subjects were given a list of the main PT501 concepts and asked to draw links between the concepts they perceived as related and to explain how they were related. The aim of this exercise was to find out how subjects grouped the concepts and to investigate their understanding of the concepts. These subjects worked through the experiment book at home, and so the above exercise was a paper and pencil exercise, included as part of the questionnaire. A related technique was used by Kelly (1955) to elicit personal constructs. The subjects were also asked to explain the concepts as though they were explaining them to someone who knew nothing about the microcomputer.

Comment sheets

In the later studies of DESMOND and Logo, protocol analysis was not the main method used. There were more subjects involved: twenty subjects were each to study the two languages, at home, in their own time (estimated to be 20 hours). It was decided, therefore, not to collect on-line protocol data. Subjects were asked to work through a practical book (as with SOLO and PT501) and were given a log book of comment sheets in which to record their attempts at all the exercises and practical work, and to give detailed comments about any problems they encountered. This would give a much more comprehensive record and would provide more quantifiable error data than for the previous studies where the subjects missed out some of the exercises.

Interviews

To capture some of the benefits of protocol collection, interviews were set up every two weeks, for pairs of students, to provide a forum for discussing any problems. There were two main reasons for interviewing pairs of students together. Firstly this technique has proved very useful in stimulating think aloud protocols (O'Malley et al, 1985), as subjects often forget to think aloud, or find it difficult to make their thoughts explicit if they are working alone. Secondly, subjects are more likely to speak about problems they are having if they know that others are experiencing similar problems.

The format of the interviews was to allow time at the beginning for subjects to talk about the problems they had had. Following this, the experimenter and subjects looked at the exercises which had been attempted, and subjects were invited to comment, explain their difficulties, and if appropriate to explain how the programs which they had written worked. Although they were retrospective, the protocols from these sessions proved to be rich and invaluable in filling in gaps in the information provided in the sheets, and in providing answers to questions such

as: "Why were you unable to complete question x? What exactly was the problem?".

Categorisation of errors and problems

The data from the DESMOND study yielded the most comprehensive number of categories of problems, and so where possible these categories were used to organise the data from all four languages. However, because the methodology used to collect the data varied between the first two studies of SOLO and PT501 and the later studies of Logo and DESMOND, and because the languages and curricula varied, the range of categories present in each set of data also varied. There are three main groups of problems:

1 Programming.

These are problems concerned with the domain itself, with understanding programs and writing programs, as opposed to the problems related to the instructional style used or affective problems

2 Instructional problems

These are to do with the style or method of instruction

3 Affective problems

These are problems such as students' attitudes towards learning to program.

Within each group there are several different types of problems but these vary across the different languages, and each chapter discusses the types of problems that were encountered in that particular environment.

Below is a brief account of the subjects, tasks and methods used for studying all

four languages, and this information is summarised in table 5.1. A more detailed description of the methods used is given in the appropriate chapters.

SOLO

The SOLO studies included two pilot studies and a main study. The first pilot study was conducted in order to find out the kinds of problems which were encountered by students, and to determine the best way of investigating such problems further. Two different methodologies were investigated: collecting data at a residential summer school, and individual case studies.

Pilot Study 1: Collecting data at summer school

Subjects were Open University cognitive psychology students using SOLO for the Artificial Intelligence project at a residential school. During the week in which the study was carried out, 45 students completed the project working in groups of between 2 and 6 students. The investigator joined one group as an observer and kept a record of the group's work, and took notes taken during debugging sessions to provide the context for later error analysis. Prediction tasks were also used where students were asked to explain what they thought was going on in their programs, and to predict their program's performance.

Pilot Study 2: Individual case studies

Two subjects worked through the SOLO manual at the Open University, and attempted the exercises and the tutor marked assignments. Think-aloud protocols were collected for part of this process. The aim of this was to investigate the feasibility of collecting detailed think-aloud protocols. The first subject, S1, was asked to read through the SOLO programming manual and complete all the exercises, and to think aloud whilst attempting the tutor marked assignment. S2 was also asked to read through the SOLO programming manual and complete all the exercises but she was asked to think aloud during the whole process. Both

subjects had a terminal available but were encouraged to work out paper and pencil solutions to exercises before trying them out on the terminal. Both the subjects were tape recorded and the recordings were later transcribed.

The Main Study

Thirteen subjects worked through SOLO. They were cognitive psychology students who were learning SOLO as part of the course, but agreed to come and work on SOLO in the laboratory using a slightly shortened version of the SOLO manual. There were two groups, the first consisting of ten subjects and the second of three subjects, who studied SOLO at different times. Group 1 was an "assisted learning" group: the experimenter came to their rescue if they had struggled too long with one topic, which ensured that an adequate range of topics was studied. Group 2 were given no help at all, so that their errors could be thoroughly recorded.

The subjects were asked to think aloud as they worked through the exercises given in the SOLO booklet, to write down their answers, and to describe what they were thinking as they attempted the problems. The interactions between the subjects and the computer were tape recorded using Cyclops, which was described above.

PT501

Study 1

16 Cognitive Psychology students were sent the PT501 home experiment kits a short while before they began studying the SOLO manual. Half of the subjects were asked to work through the PT501 book before they worked through SOLO, and half were asked to wait until they had finished working through the SOLO manual. They were asked to work through the experiments in the booklet, answer all the in-text questions and write their answers on the experiment booklet. Subjects also commented on the booklet itself. They worked on the experiment

book at home and in their own time. The subjects also filled in an initial questionnaire on their attitudes to the role of AI in the course, and to the use of computers more generally. Final questionnaires were sent to the subjects two months after they had finished working through the experiment book and concept interviews (Appendix 7.1) were tape recorded at the university, and consisted of asking the subjects to describe, in their own words, what each of the concepts meant to them.

Study 2

Three subjects were recruited to work through PT501 and SOLO. These are the same subjects who were Group 2 for the SOLO study. One spent all his time working on SOLO and never began the work on PT501, and so only 2 subjects studied PT501. They were asked to think aloud as they worked through the exercises in the instruction booklet, to write their answers down, and to describe what they were thinking as they tried to solve the problems set in the booklet. The experimenter only helped the subjects if they were unable to move on without help. Otherwise the subjects were prompted to explain their problem, but not helped to solve it, although as time passed they might be pointed in the right direction.

DESMOND and Logo

Most of the details of the subjects and procedures used in these 2 studies has already been given. Twenty subjects formed two groups of 10: the first group worked through DESMOND and then Logo, and the second group worked through Logo and then DESMOND. They completed all the in-text exercises, filled in comment sheets and attended interviews, as described above.

The information about subjects, tasks and the data collected is summarised below:

	SOLO			PT501		LOGO	DESMOND
	Pilot study 1	Pilot study 2	Main study	Study 1	Study 2		
Subjs.	D303 students (Total gp of 45)	2 (non-D303)	13 Ss in 2 gps Gp 1 (10) Gp 2 (3)	16 D303 students	2 paid subjects (from Gp 2)	20	20
Task	Summer school project	Work through SOLO manual, and complete exercises. S1 to complete assignment		Work on PT501 exp book at home	Work on PT501 exp bk in lab.	Work through LOGO and DESMOND. Complete all in text exercises. Fill in comment sheets. Attend interviews	
Data	Traces of work. Record of debugging session. Errors. Observation notes	Taped transcripts of think-aloud protocols. In main study these include synchronised record of key presses		Annotated exp. book Q'nnaire Concept interviews	Taped transcripts of protocols	Comment sheets, including attempts at programming and other exercises. Interview transcripts.	

Table 5.1: Subjects, tasks and data

5.4 CONCLUSIONS

This chapter has discussed the rationale for the methodologies used, and argued that the appropriate approach was a qualitative one, where data is obtained without being overly constrained by theory, and is categorised so as to derive theory from it. However the later studies benefited from using both qualitative and quantitative methods. Evidence for the validity of one technique which is used in this study, protocol analysis, was examined, and recommendations given for its use. An overview of the subjects used and the methods adopted across the different studies

has also been given.

Chapter 6

LEARNING TO USE SOLO

Contents

6.1 INTRODUCTION

The aim of this research was to investigate the difficulties which novices encounter when learning a programming language which is taught via a conceptual model, - and when the learners are learning at a distance. In chapter 4 the programming languages were described, and were presented from the point of view of the designers and teachers, and in particular their conceptual models were analysed and discussed. This analysis of the conceptual model suggested a "rating" for each of the languages. The kinds of problems that learners experience in learning these languages however, requires empirical investigation. The next four chapters present the same languages from the learners' point of view, and report on empirical studies of learning the languages.

In order to do this, however, it was necessary to research the feasibility of the proposed approach, and to investigate how the conceptual models were perceived by the learners and whether the learners experienced the kinds of problems discussed in chapter 2. The first part of this chapter describes preliminary investigations into methods and techniques. The aims of the thesis as reported in previous chapters require detailed investigations of the difficulties faced by individual learners of real programming languages making many of the more traditional methods in this area unsuitable. For instance, time-and-error scores have been used (Gilmore and Green, 1985) to compare the comprehensibility of different languages, but they do not reveal the sources of difficulty. The rationale for the methodologies adopted in this thesis was discussed in chapter 5, and protocol collection, the main methodology used in the studies, was described and evaluated. The next three sections of this chapter (sections 6.2 - 6.4) describe pilot studies used to evaluate candidate methods for the empirical studies.

All the studies reported in this chapter are of students learning to use SOLO, which has already been described in chapter 4. It is part of a third level cognitive psychology course and was being used in a project at a residential summer school which enabled some of the pilot studies to be carried out during this period.

A further objective of the study was to investigate the transfer effects of learning a high and low language in sequence. Low level languages such as the assembler learnt in the PT501 course discussed in chapter 4 have the virtue of being close to the machine, whereas high level languages are closer to the problem. Learning two languages at different levels should help novices form a more accurate model of low level concepts (such as variable) by giving them a high level view and a view which is closer to the machine and fills in the details. Weyer and Cannarra's transfer study (Weyer and Cannarra, 1975) was discussed in chapter 2, and although they had some success in teaching a high and low level language concurrently, it was concluded that it remains open whether learning a low and high level language sequentially facilitates learning. It was decided, therefore, to take a naturally existing population of novices who had learnt (or were learning) SOLO and to expose them to a second, low level language (PT501). The SOLO study reported in the second part of this chapter is part of this larger study, which will be reported in chapter 7.

6.2 COLLECTING DATA FROM SUMMER SCHOOL: PILOT STUDY 1

The pilot study was conducted in order to

1 find out what kinds of problems were encountered by students, and

2 to determine the best way of investigating such problems further.

Two different methodologies were investigated: collecting data at a residential summer school, which is reported in this section; and individual case studies, reported in section 6.3.

Subjects

Subjects were Open University students attending the cognitive psychology's week long residential school, which is a compulsory component of the course. Two groups of between 20 and 30 students complete the artificial intelligence (AI) project: one group during the first half of the week, and one group in the second half of the week. During the week in which the study was carried out, 45 students completed the AI project. These students are studying for their degree part-time, and are mature students who have undertaken at least one previous psychology course.

Task

Each group chooses a project from among the following: propagating inferences; a simulation of a model of long term memory, children's arithmetic errors and schema matching. Further information on these projects is given in Appendix 6.1. The project is done in groups of between 2 and 6 students. Each project has a "trailer" to give some information about the project and an idea of what's possible, which is run before the project starts. Each trailer is about 2 hours.

Methodology

Information was collected by:

1 Observation and analysis of errors collected in records of the students' work. The investigator joined one group during the first half of the week and followed their project through. None of this group had any programming experience. A record of the group's work (and of others) was kept by using double paper with interleaved carbon instead of the usual printout paper, and notes were taken during debugging sessions to provide the context for later error analysis. One debugging session was tape recorded.

2 Prediction tasks where students were asked to explain what they thought was going

on in their programs, especially bugged ones, and to predict their program's performance.

3 A debriefing session at the end of the project.

4 Informal and tape recorded interviews to investigate affective issues.

Results

Two broadly different types of error were identified in SOLO programs in the pilot study.

The first type were errors in the program either corrected automatically (which in itself caused problems at times) or which led to error messages. These may have been "trivial" errors or manifestations of underlying misconceptions about how SOLO works. The first type include some of the same errors as those noted by Lewis (1980). One example is attempting to "describe" the wrong end of a triple, i.e. typing:

DESCRIBE FLEAS

in an attempt to retrieve the triple

FIDO HAS FLEAS

Other examples include confusing "*" with "?", trying to 'forget' a procedure, and to 'list' nodes. Two of the examples above can be related to SOLO's lack of consistency: trying to 'forget ' procedures is quite sensible, as nodes can be forgotten, and listing nodes may be sensible, as procedures can be listed. This (early) version of SOLO also violates the 'show users changes' rule by removing parameter slashes when the parameter is not included in the title of the procedure, and not showing the change to the student, e.g.:

SOLO: EDIT ADDTIJ
.....: 3 NOTE /X/ IS TRUEBLUE
.....: DONE
OK....I have re-defined how to 'ADDTIJ'

Here the student is amending line 3 of ADDTIJ. The student's input is underlined and followed by SOLO's response, DONE, indicating that the change has been completed. The student now wants to see the amended procedure and so types LIST ADDTIJ and the procedure is displayed:

```
SOLO: LIST ADDTIJ
1    FORGET TORYSCORE  IS ?
2    NOTE TORYSCORE  IS 6
3    NOTE X IS TRUEBLUE
```

Because the parameter is not given in the title line, SOLO removes the slashes in line 3 but does not inform the user that this has happened, or explain why. The effect is to treat X as a literal and not as a parameter name. On listing the procedure the user sees that the edit has not produced the desired effect and tries again. This particular problem occurred several times and was noted by Lewis (Lewis, op. cit.) in his analysis. Messages such as "ooops.....the variable *A has no value (no 'contents') at this point" were quite common, and occurred because in writing independent procedures where values got passed down from one to another, values did not always get passed on in the way that students assumed. This could be due to a number of reasons: for example that they were not skilled at passing values, they lost track of the contents of a variable, or assumed local variables to be global.

The second type of error did not appear as a 'bug' but included programs which did not work as the user intended or contained unhelpful redundancies. Despite forcing users to use both branches of the conditional, some users may attend to one branch only: for example, CONTINUE may be used inappropriately, when in fact EXIT is required, as in the example below where the control statement in line 20A should be EXIT:

```
ADD1/NODE/

10 CHECK/NODE/ IS ?A
    10B IF PRESENT: FORGET /NODE/ IS *; CONTINUE
    10B IF ABSENT: CONTINUE
20 CHECK *A PLUS1 ?B
    20A IF PRESENT: NOTE /NODE/ IS *B: CONTINUE
    20B IF ABSENT: PRINT "IT'S A MESSUP";EXIT
```

There are several possible interpretations of this error:

1 A logic error - or perhaps an assumption that it didn't matter: it would be safer to have a CONTINUE.

2 Using language with everyday connotations may lead to misunderstanding: e.g. CONTINUE may be taken to mean "continue searching through data", or a confusion of levels where CONTINUE means "continue with text" (as in "continue on page 81") rather than 'continue flow of control'.

3 CONTINUE could be taken from an example given in the manual.

Such "bugs" may indicate underlying confusion which is both interesting and informative.

In another example, the students spoke of the procedure 'matching' two nodes:

```
TO MATCH /X/ /Y/
1   CHECK /X/ FAVOURS ?DENAT
    1A IF PRESENT: CONTINUE
    1B IF ABSENT: PRINT ""/X/" IS NOT A TORY'; EXIT
2   CHECK /X/ LOOKSFOR ENTERPRISE
    2A IF PRESENT: CONTINUE
    2B IF ABSENT; WAIT /X/; CONTINUE
3   CHECK /X/ NEEDS WEALTH
    3A IF PRESENT; CONTINUE
    3B IF ABSENT; WAIT /X/: CONTINUE
4 PRINT ""/X/" IS A TORY": CONTINUE
```

This procedure does not "match" anything (in the sense of "compare"). Although it has two parameters it does not refer to the second in the body of the procedure. According to the students it was originally intended to "compare" two things: does it fail because they discovered that this was not, in fact what was required? Or because the name was taken from the demonstration procedure and they don't know how to go about the matching? Notice also, that this program finishes with a CONTINUE, whereas it should, of course, be EXIT.

Literature on children learning LOGO (Noss, 1985) suggests that goals are changed when they can't be satisfied: it's possible that here, the students didn't know how to satisfy their original intention using SOLO, or that their original intention wasn't particularly clear. It is common for SOLO students (and no doubt others), to think that they have analysed a problem thoroughly, and know just how to go about it, and they fail to understand a tutor's questioning on exactly how they will represent X or Y until they have to program it when they discover that their plan is not sufficiently explicit.

Discussion

The first aim of the pilot study was to see whether students had problems in learning SOLO, and if so, what they were. They do have problems and two kinds of error have been identified: the first kind were identified by SOLO as errors, whereas the second type were not bugs, but a mismatch between the learner's intentions and the program which was written. The second aim concerned the feasibility of the methodology. Using a range of methodologies: particularly the combination of observation and error analysis was on the whole successful. However, one of the problems in studying SOLO learners in their groups at summer school was that it was very difficult to distinguish individual misconceptions, and the path of an individual's learning, given that the work was done in a group, and one individual may take over a program and alter it, or a particular person may be influential (and wrong) about what should

happen etc.

The next section discusses the evaluation of a methodology which overcomes this problem by adopting an individual case study approach.

6.3 A CASE STUDY OF PROBLEM SOLVING IN SOLO: PILOT STUDY 2

Two subjects worked through the SOLO manual at the Open University, and attempted the exercises and the tutor marked assignments. Think-aloud protocols were collected for part of this process. The aim of this was to investigate the feasibility of collecting detailed think-aloud protocols.

Subjects

One subject, S1, was a student between school and university who was working as a temporary clerk at the Open University during his vacation. He had some programming experience in BASIC. The other, S2, was a research student working at the university. She had a psychology degree and no programming experience.

Task

S1 was asked to read through the SOLO programming manual and complete all the exercises, and to think aloud whilst attempting the tutor marked assignment. S2 was also asked to read through the SOLO programming manual and complete all the exercises but she was asked to think aloud during the whole process. Both subjects had a terminal available but were encouraged to work out paper and pencil solutions to exercises before trying them out on the terminal.

Methodology

The subjects were tape recorded and the recordings were later transcribed.

Results

Subject 1

The first subject's procedures and protocols showed evidence of two confusions. Firstly he was confused about the use of variables and parameters. The fact that he wasn't clear about the distinction between stars and slashes (syntactic markers for variables and parameters) is evident from the second procedure he defined (CATCHBOSSOF) where he could not decide whether the title line should be CATCHBOSSOF /X/ or CATCHBOSSOF *, and in the recursive call in line 2, he wandered whether he could use the variable. His protocol also showed evidence of this uncertainty. The second confusion was between defining and calling a procedure.

It is not clear how distinct these confusions are: one may exist only as a result of the other, or they may be related. In order to attempt to untangle such confusions, more context is necessary. In this case, only the subject's attempts at the tutor marked assignment had been recorded. What was required was more information on the kinds of mental models that he had developed of variables and parameters and of defining and calling a procedure. This information could be gleaned by taking think aloud protocols of subjects as they encountered these concepts in the text. This was done with the second subject.

Subject 2

S2 thought aloud as she read through the manual. Although she had been encouraged to think through the solutions before trying them out on the computer she developed into a novice hacker! A protocol extract is given below of S2 working through an early part of the manual. The convention for this and all subsequent SOLO protocol extracts is as follows. The subject's protocol is broken into numbered segments so that they can be referred to in the discussion of the protocol. The numbering also indicates where parts of the protocol have been left out of an extract, and dots (e.g. "carry it....to

the routine here") indicate the omission of smaller segments (often asides). Interjections by the experimenter are marked [E] and unnumbered. Everything the subject types is given in bold. The other typing, e.g. the SOLO: prompt is given by the SOLO program:

	S2's protocol	**What appeared on the terminal**
04	OK, ...to grumble	SOLO:
05	and in inverted commas, too good today	**.10 "I DON'T FEEL TOO GOOD TODAY"**
06	System error, not your fault, please try your input again	System error, not your fault, please try your input again
07	OK, I'm just going to put grumble let's see if.. we're going to have to redefine our procedure or not, - grumble is an undefined procedure yes, that's right, grumble has not got a definition, ... so, to grumble, return	SOLO: GRUMBLE SOLO does not have a procedure called GRUMBLE. Do you want to define one? SOLO: TO GRUMBLE
08	Right. Ah! Ah! OK! We're not writing print! We're not writing print! Every time the word grumble comes up as the opening gambit of a procedure, um, what is printed, will actually.. We've had ... the old system error here but actually it was my own fault because I didn't write print prior to that..	**..10: PRINT " I DON'T FEEL SO GOOD TODAY"**
11	To Foo, right so we're going to print	**..10 PRINT "BAZ GLUB FOZ"** **..20 DONE**
12	So that is.. how do I explain to foo, it's what you're telling ..um.. you're defining the procedure rather than asking for that procedure to be laid out, you're defining the procedure to foo, and I'm going to print, no, I'm going to note. What's the difference between print and note here?	
13	OK I'm going to put print here just to find out what the difference is between print and note in this respect	FOO has been successfully defined TO FOO 10 PRINT BAZ GLUB FOZ SOLO: TO FOO
		You have already got a procedure called FOO. Do you want to erase the existing procedure FOO and replace it with a new one?
14	Do I want to erase the old..?	
15	No,...so, note BAZ,BIF,ZAP	SOLO: NOTE BAZ BIF ZAP
16	OK BAZ BIF ZAP	OK BAZ └–>BIF–>ZAP
17	No. I've done it wrong, I haven't specified that we're still into defining the procedure FOO	SOLO: TO FOO

18	OK note	You have already got a procedure called FOO. Do you want to erase the existing procedure FOO and replace it with a new one?
19	I want to keep it	
20	OK I'm going to put, now I'm confused here between the meaning of print and note, to grumble, er.. you print the three things..er the three descriptions and then when it's printed up, grumble, the three things come out without the print	
21	OK let's to zip, so	SOLO: **TO ZIP**
22	Print	**...10: PRINT "ZAP ZOP ZIP"** **...20: DONE**
23	Oh I see, so now I just put zip and I get these three. Oh, that's good. But it still doesn't help me with the note: what's the difference between the print and the note?	SOLO: **ZIP** ZAP ZOP ZIP

Protocol extract 6.1: S2 working through SOLO manual

This data is much richer. Some of the subject's misconceptions are much clearer - for example she initially has problems because she omits the print statement. In segment 12 she talks about her interpretation of TO FOO: "it's what you're telling, ...you're defining the procedure rather than asking for that procedure to be laid out". Later she is uncertain about the difference between note and print and decides to carry out an experiment to clarify the difference. She then gets caught up in the operational activities of carrying this out, - but it can be assumed that she still has the goal of understanding the difference between print and note. In fact, she was not able to resolve this herself and asked the experimenter, much later.

6.4 DISCUSSION OF THE PILOT STUDIES

Two different methods of investigating students' mental models were evaluated. The first was to collect data from a number of students, concentrating on the programs they had written at summer school and combining observation and error analysis. This worked well, as long as there was sufficient context to analyse programs. At summer school this was often not the case as students were working in groups and it was often not clear who had contributed to a particular program.

The second method overcame this problem by focussing on individuals, and collecting think-aloud protocols from students as they read through the SOLO manual and carried out various tasks. The protocols of the first subject attempting the tutor marked assignment revealed some misunderstandings. To understand these further it would have been necessary to have collected more protocols from this subject, particularly his earlier attempts to understand and use the concepts that he had difficulty with.

In order to gain insights about a subject's mental model, it is necessary to gather detailed information about her perception and understanding while she is engaged in the learning process. The "snapshot" of the first subject was interesting, and provided quite a lot of information, but was not sufficient, whilst the more extensive protocols gathered from the second subject provided the more detailed process data. However, it should be acknowledged that whilst this is a rich source of data it is also expensive and time consuming to collect and analyse.

It was decided, therefore, not to pursue the idea of collecting data at summer school but to follow individuals as they learnt SOLO, and the results of this investigation are reported below in the second part of this chapter.

The next section describes the subjects and outlines the task that they undertook, and the methodology. In section 6.6, the subjects' difficulties are discussed: in particular difficulties in understanding control statements, writing their own programs and problems caused by the subjects constructing interpretations. These results are discussed in section 6.7.

6.5 AN EMPIRICAL STUDY OF NOVICES LEARNING SOLO: SUBJECTS, TASK AND METHOD

The next four sections describe an investigation into the difficulties experienced by novices learning SOLO. A small number of studies have already been carried out on SOLO, and these were discussed in chapter 2. Lewis (1980) and Eisenstadt and Lewis (1985) analysed a large number of programs written by students and categorised the errors. Although they attempted to identify the causes of the errors from the context, these studies did not focus on the difficulties experienced by individual students. Kahney's study (1982) looked specifically at problem solving behaviour, including novices' models of recursion. He did not, however, investigate the kinds of difficulties students may have had very early on with conditional structures which may have led to their impoverished models of recursion. This study therefore focussed on the process of acquiring expertise, and on the problems that novices have in developing their skills as they work through the curriculum.

Thirteen subjects worked through SOLO. They were students who were studying the cognitive psychology course, and would therefore be learning SOLO as part of the course, but agreed to come and work on SOLO in the laboratory using a slightly shortened version of the programming manual used on the course. The 13 subjects form two groups, group 1 consisting of ten subjects and group 2 consisting of three subjects, who studied SOLO at different times. Group 1 was an "assisted learning"

group - the experimenter came to their rescue if they had struggled too long with one topic. This was necessary to ensure that an adequate range of topics was studied. A smaller number of students, group 2, were given no help at all, so that their errors could be thoroughly recorded. The experimenter started the session in the room with the subjects to prompt them to think aloud, and then sat in an adjacent room to be "on call" should the subject feel totally stuck. The experimenter only helped the subjects solve the problem they were working on if they were unable to move on without help. Otherwise the subjects were prompted to explain their problem, although if they were really struggling they were pointed in the right direction.

Subjects were asked to think aloud as they worked through the exercises given in the instruction booklet, to write down their answers, and to describe what they were thinking as they attempted the problems set, although one subject had already worked through the manual as part of the course and came to the laboratory to talk about the work she'd done and to complete further exercises. The interactions between the subjects and the computer were tape recorded using Cyclops, a device developed at the Open University enabling keystroke and voice data to be recorded simultaneously (Scanlon and O'Shea, 1987) .

6.6 RESULTS

Most of the data is qualitative rather than quantitative. Also, a lot of the exercises in the manual are not programming exercises but exercises which lead students towards writing programs such as tracing the flow of control of given programs. Examples of these exercises are given in Appendix 6.2. The next section discusses the problems that the subjects had, and is based on their verbal and terminal protocols. Generalisations are offered about the problems the subjects experienced, and these are illustrated with protocol extracts. This is an inductive approach in that generalisations are inferred

from multiple examples and the most typical examples are used to illustrate the generalisations. There is more data available for the second group than for the first group, as they were left to struggle when things went wrong, and so the detailed examples are often drawn from this group. Many of the SOLO problems are domain related, i.e. they are programming problems and these are discussed first. Some of the subjects' problems however were instructional in that they were related to their approach to learning and these will be discussed later.

Understanding programs

1) Flow of control

Many of the subjects had problems in understanding the SOLO control statements. Ten of the 13 subjects had problems in understanding the effect of CONTINUE and EXIT: i.e. what happens to the flow of control following a CONTINUE or an EXIT statement. For example, four of the subjects could not predict the outcome of JUDGE given the definition in figure 6.1 and the database in figure 6. 2.

```
TO JUDGE /X/
1   PRINT "HERE'S WHAT I THINK OF" /X/
2   CHECK /X/ VOTES INDEPENDENT
2A  IF PRESENT: PRINT /X/ "IS A FREE THINKER"; CONTINUE
2B  IF ABSENT: PRINT /X/ "IS A PUPPET"; CONTINUE
3   PRINT "THAT'S ALL I HAVE TO SAY"
```

Figure 6.1: The judge procedure

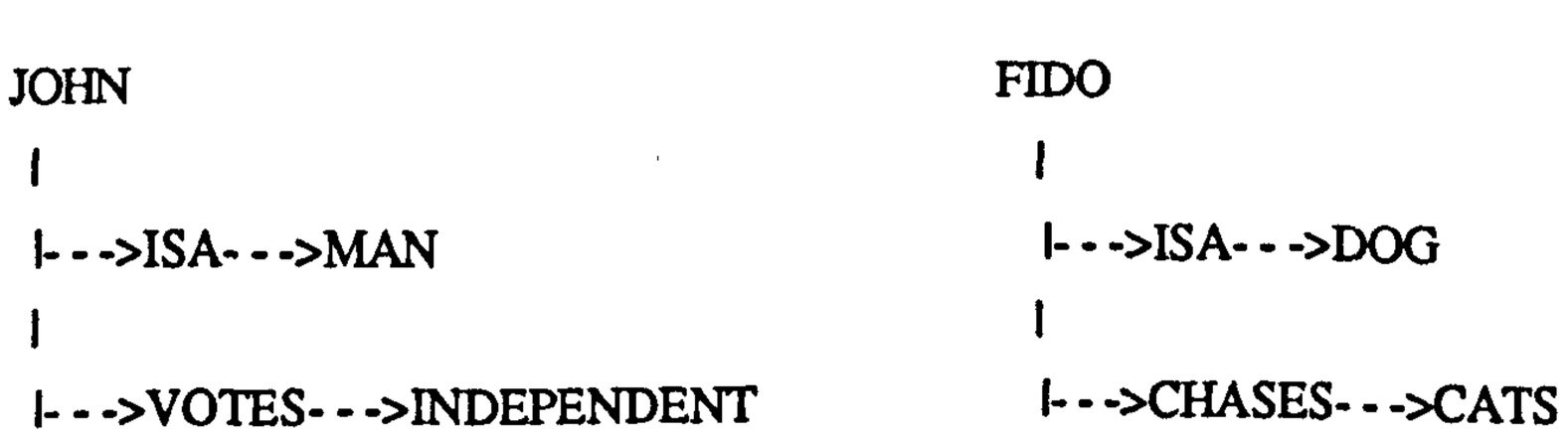


Figure 6.2: The database used by the JUDGE procedure

One subject predicted that the outcome to JUDGE FIDO would be "that's all I have to say", whereas what is in fact printed is:

"HERE'S WHAT I THINK OF FIDO

THAT'S ALL I HAVE TO SAY"

and another subject predicted that JUDGE JOHN would produce

JOHN VOTES INDEPENDENT

even though the procedure does not contain the line

DESCRIBE JOHN

which would print this.

These four subjects did not know where flow of control statements should be used, and in discussing how to write procedures such as JUDGE or ASSESS (discussed later) made comments such as: "Do I need a CONTINUE or what?" (after writing the title line of JUDGE), and: "I don't need a CONTINUE now do I?" after IF PRESENT. Other subjects demonstrated quite gross misunderstandings of control statements. One of the subjects explained the use of CONTINUE as follows:

"CONTINUE is more or less to keep the machine on the boil, it keeps it going."

She didn't, however, know <u>what</u> was kept going. Another subject had no idea of when CONTINUE should be used, and why it was not needed in the CHECK statement.

<u>2) Inference limitations</u>

Another type of problem is that subjects developed erroneous beliefs about the particular procedures which are used in the text to introduce control statements. They were confused about how procedures work: what the scope of a procedure is, and how it uses the triples in the data base. Almost half of the subjects (6/13) made predictions about the behaviour of the procedures which suggested that they believed that the procedures had access to data in addition to that in the data-base, and that the procedures could make inferences that had not been programmed. An example of this is given below, which is an extract from a protocol of a subject talking about the

JUDGE procedure. He has just typed JUDGE ALICE, and received the response:

ALICE IS A PUPPET:

1 I don't know why it's got to the conclusion that Alice is a puppet.....because she is not...

[E] Try stepping through the procedure

2 Oh, I see, regardless of whatever else is in there, if it hasn't got Alice votes independent it will say Alice is a puppet,...so it would say the same for dog?

Protocol extract 6.2: A subject talking about JUDGE

This person initially believed that only **positive** information could affect which branch a procedure took, i.e. which triples were in the data base, but then came to realise that the absence of triples was equally important. Another subject said:

"Judge means to me, judge what he is or what he does, judge what he is, 'judge john is a man'".

in order to explain her prediction that JUDGE JOHN (see figure 6.1) would produce JOHN ISA MAN. This explanation of JUDGE does not relate to JUDGE and how triples are matched, but to a a belief about how some general "judge" procedure might work. This model of "judge" uses information that is in the data base (JOHN ISA MAN), but which the procedure JUDGE will not check as it neither contains the lines CHECK JOHN ISA MAN nor DESCRIBE JOHN. Another subject explained how she viewed judge as a prediction about the way someone votes, as shown in the protocol extract 6.3 below

1 To me it seems very abstract, so I try to apply it to something, - so this procedure, I see it as somebody predicting votes

[E] I see, well you could do that, - but you'd have to write it into the data base that so and so votes independent for everyone who did and then you could use judge over and over for all of them.....

3 Yes, I see. I'm assuming all sorts of information is already there.....

Protocol extract 6.3 One subject's view of judge

This person was also confused about how the procedure actually uses the database, and

seems to believe that the choice of where a line branches to is achieved by editing:

"If we were checking votes and we wanted to know how many people we would manipulate, we would delete 1A, if present."

This extract suggests a belief that the procedure or computer has access to information beyond what has been given. A similar belief is seen later when the same subject has just defined the ASSESS procedure, and says:

"If I type in assess and the program ... er... drinking beer is unhealthy or drinking whisky is unhealthy and then writes Mary drinks whisky, will it come up with Mary is unhealthy?"

Here, a procedure is expected to make an inference that has not been programmed. The person does not know what the scope of a procedure is. The three subjects who formed group 2 spent some time tracing through the execution of the procedures STRONGASSESS and WEAKASSESS, which are given in figure 6.3 along with the text which precedes them. These procedures are the closest that the manual comes to giving SOLO "plans". (As was seen in chapter 4, SOLO's conceptual model does **not** emphasise a functional view - which would help students work out and understand the SOLO plans). Taken together, the STRONGASSESS and WEAKASSESS procedures illustrate how SOLO can be used to trace through a decision tree such as those given in figures 6.5a and 6.5c. Figures 6.5b and 6.5d show how the example could be developed further to show the relationship between the flow of information and SOLO's control statements.

Two of these subjects also interpreted the STRONGASSESS and WEAKASSESS procedures in a way which went beyond the information given: attributing their own goals to them which led to expectations about how they work. This interacts with their problems about understanding control statements. The exercise which the subjects are

attempting is given in figure 6.4, and an extract from a protocol of a subject tracing the execution of these procedures is given in protocol extract 6.4.

Complicated sequences of CHECKs within CHECKs can be achieved in a single procedure by arranging your EXITs and CONTINUEs properly. For example, look carefully at the difference between the following two programs. Both are designed to 'assess' someone's physical fitness by deciding whether to print FIT or UNFIT, on the basis of certain known 'facts' (such as whether a person plays squash, rides bicycles, etc.). The first procedure is a 'weak' assessment, while the second one is a 'strict' assessement, hence the names WEAKASSESS and STRICTASSESS. Notice how the two procedures differ in their usage of CONTINUE and EXIT.

```
TO WEAKASSESS /X/
    1 CHECK /X/ PLAYS SQUASH
      1A  IF PRESENT: PRINT "FIT"; EXIT
      1B  IF ABSENT: CONTINUE
    2 CHECK /X/ RIDES BICYCLES
      2A IF PRESENT: PRINT "FIT"; EXIT
      2B 1B IF ABSENT: CONTINUE
    3 CHECK /X/ CLIMBS MOUNTAINS
      3A IF PRESENT: PRINT "FIT"; EXIT
      3B IF ABSENT: CONTINUE
    4 PRINT "UNFIT"
```

```
TO  STRICTASSESS /X/
1 CHECK /X/ PLAYS SQUASH
    1A IF PRESENT: CONTINUE
    1B IF ABSENT: PRINT "UNFIT"; EXIT
2 CHECK /X/ RIDES BICYCLES
    2A IF PRESENT: CONTINUE
    2B IF ABSENT: PRINT "UNFIT"; EXIT
3 CHECK /X/ CLIMBS MOUNTAINS
    3A IF PRESENT: CONTINUE
    3B IF ABSENT: PRINT "UNFIT"; EXIT
    4 PRINT "FIT"
```

Figure 6.3: The WEAKASSESS and STRONGASSESS procedures

Suppose that SOLO's data base contained the following descriptions:

```
FRED
 |
 |- - ->ISA- - ->MAN
 |
 |- - ->PLAYS- - ->SQUASH
 |
 |- - ->RIDES- - ->BICYCLES
 |
 |- - ->CLIMBS- - ->MOUNTAINS
```

```
MARY
 |
 |- - ->ISA- - ->WOMAN
 |
 |- - ->PLAYS- - ->SQUASH
 |
 |- - ->RIDES- - ->BICYCLES
```

How would SOLO respond to each of the following (be sure to work through the procedures step by step, and follow the control statements precisely):

(a) SOLO:WEAKASSESS FRED
(b) SOLO:WEAKASSESS MARY
(c) SOLO:STRICTASSESS FRED
(d) SOLO:STRICTASSESS MARY

Figure 6.4: The question set on WEAKASSESS and STRONGASSESS

1 ...So, check Mary plays squash, yes Mary does.., if present print fit exit and that's it, finished.

2 So it doesn't matter that Mary also rides bicycles as well and that she doesn't climb mountains, but is the actual procedure going to stop there if it exits there?

3 .....O.K. So it won't. The fact that she does or doesn't ride bicycles or does or doesn't climb mountains won't come into the assessment of her being fit or not?

[E] No, that's right. .....So what were you worried about, were you not sure what the exit did there?

5 I suppose so, yes, I suppose I was just wondering if it... the exit just meant stopping on that line... or going on to the next one, but obviously it doesn't because that's what continue means.
[E] What, stopping printing that, but then having a look at the other as well?

6 Yeah,... does it sort of look at print fit and then go on to the next one and have a look at that one... it doesn't do that.

Protocol extract 6.4: S13 tracing through WEAKASSESS

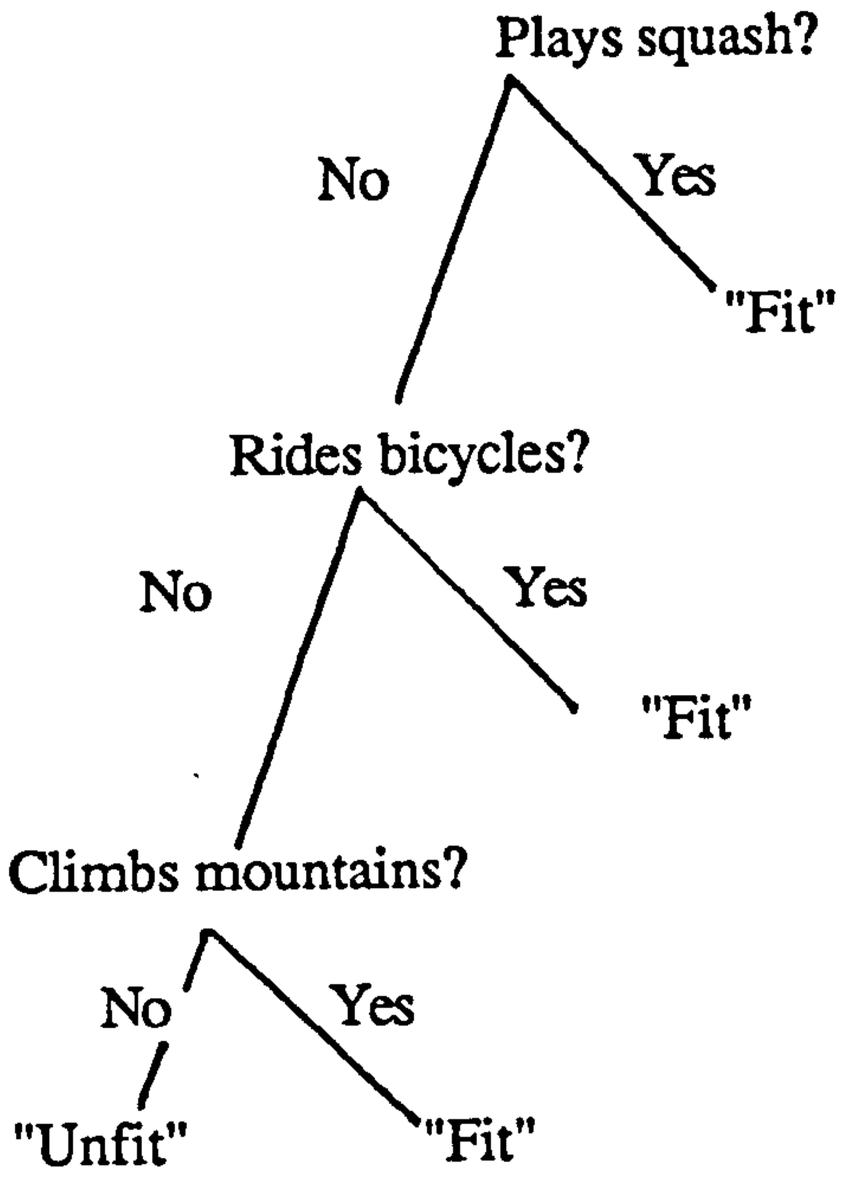


Fig 6.5a

Decision tree for WEAKASSESS

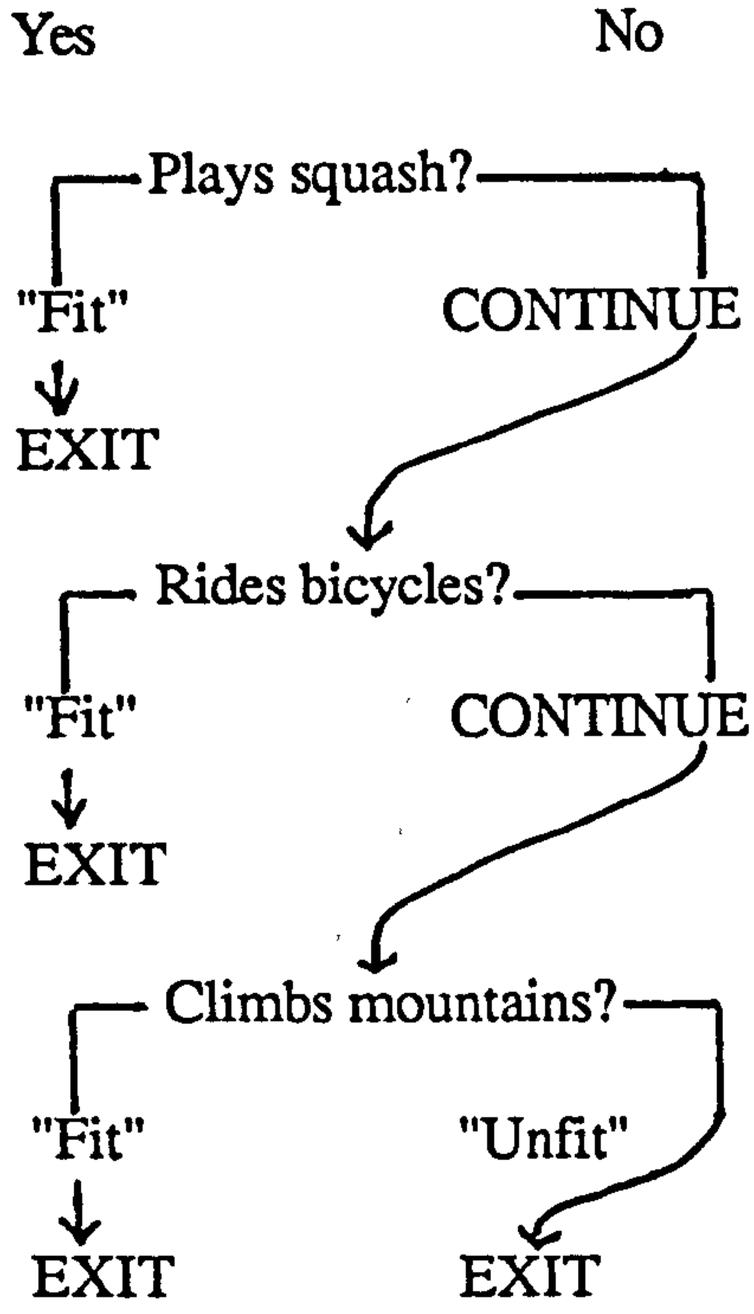


Fig 6.5b Chain diagram showing flow of control in WEAKASSESS

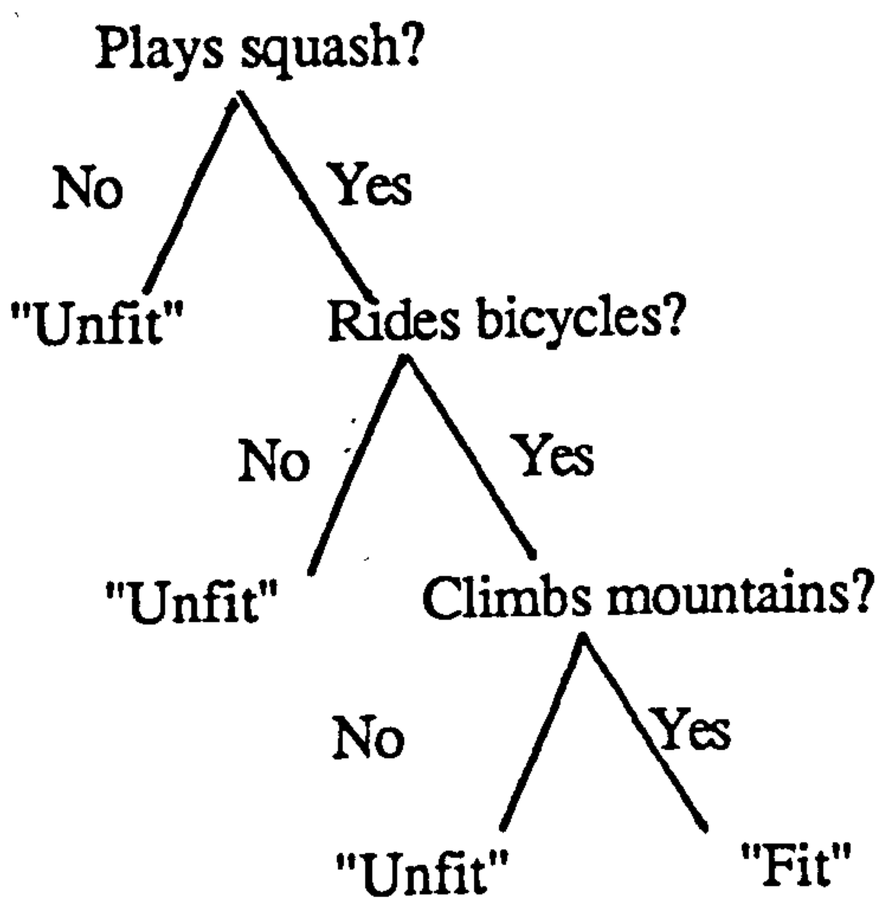


Fig 6.5c

Decision tree for STRONGASSESS

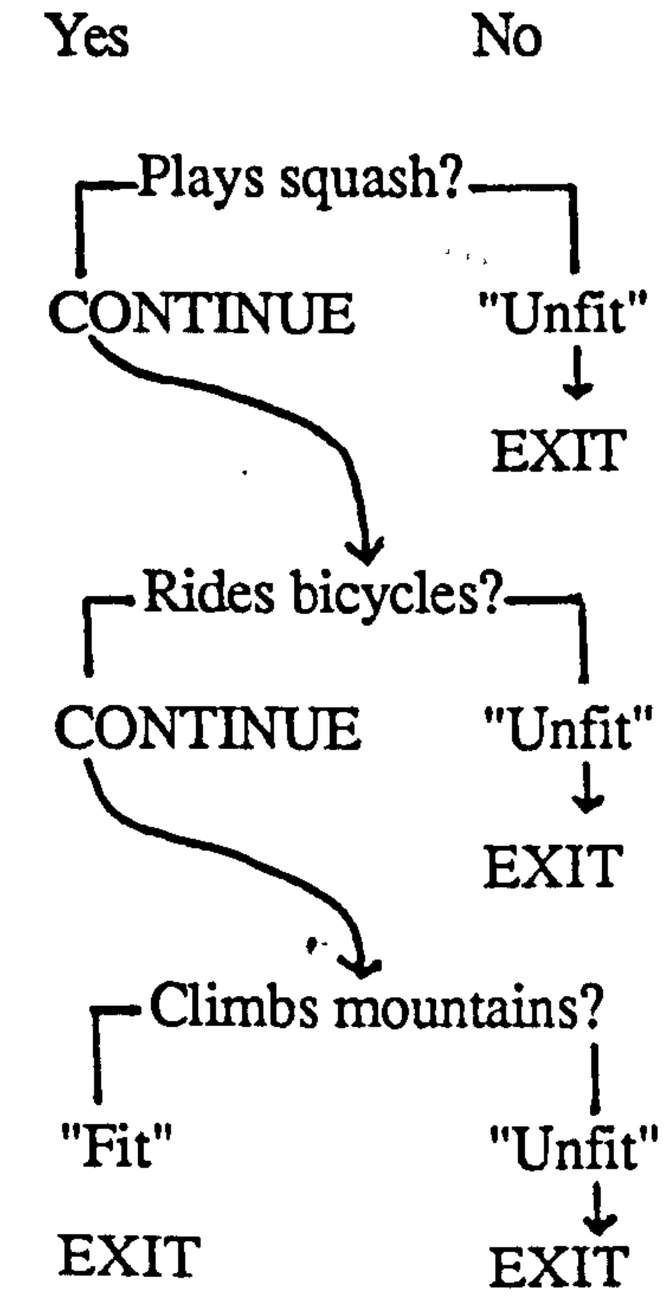


Fig 6.5d: Chain diagram showing flow of control for STRONGASSESS

In extract 6.4, S13 is not clear about the difference between the two control statements which have just been introduced (segment 5 of the extract). This is quite common at this point - many of the protocols show that CONTINUE particularly is not well understood. Earlier (2,3) he is uneasy about the redundancy in WEAKASSESS: the procedure is not checking all the information available. Part of the problem is his belief that the WEAKASSESS procedure should take account of all the available information. This leads to conflicting hypotheses about the meaning of EXIT: he believes that EXIT halts the procedure, yet if the procedure halts at this point (line 1A), the triple MARY RIDES BICYCLES will not be checked, and this conflicts with his expectation about how the procedure works. This leads to a hypothesis (6) that even though the procedure has halted, it may be able to 'have a sneaky look' at the rest of the data. A similar worry about WEAKASSESS is illustrated in the extract below:

1 Well ...where it says Fred plays squash, which, if present, print fit exit, so would that exit not bother with those checks as to whether he rides bicycles and climbs mountains?

[E] Well what do you think it would do?

2 I think it wouldn't. I don't think it would..because it would only continue for the next check if the first was absent, so it would stop there.....

3 That surely, it's a bit, why can't you check though all of them?

[E] I'm not sure what your question is.. it's a weakassess, right? So you're saying that it would exit there?

4 Only after it had checked that he plays squash but as he rides bicycles and climbs mountains, is there no way of checking all that in the same thing, to include it all?

[E] Mmm,... it just doesn't do it.

5 But it wouldn't do it in that particular thing. That one would do the same thing, it would ignore Mary isa woman and it would go on to Mary plays squash, that would be present so it would print fit and that would be it. And then strictassess Fred, so.....I see, yes.....so.

[E] Does that make it clearer... what you were asking before?

6 Yes. That would check whether he plays squash. That would be present so it would continue to bicycles, ..I see yeah and that would continue through the whole lot and the same with Mary, yeah.

Protocol extract 6.5: S14 working on weakassess

Extract 6.5 shows the subject's unease about the EXIT 'ignoring' the triples about

riding bicycles and climbing mountains (1) but the next segment (2) indicates that she understands what exit will do here but has an expectation about the function of the procedure, i.e. that it behaves like STRICTASSESS. This leads to confusion about the control statements: if the program works through the data, then exit could not, at line 1A, end the program. She expects WEAKASSESS to behave like STRICTASSESS which is given later in the text (see figure 6.3). Two different mental models are active here, and are in competition. One of the program's function which is misleading, and one of EXIT, which is correct.

Writing programs: the ASSESS problem

Many of the problems experienced by the subjects are evident in their attempts to write their first procedure. The problem set is given in figure 6.6. below:

Define your own procedure called ASSESS which prints out UNHEALTHY if someone (the node to which it is applied) either drinks whisky, on the one hand; or else if that person both smokes cigarettes and drinks beer.

Using the NOTE procedure, add some descriptions of your own to SOLO's database, and try out your ASSESS procedure to get it working properly. You must decide for yourself how you are going to represent "drinks whisky" etc. in the database.

Figure 6.6: The ASSESS problem

This problem is the first that requires the learner to produce their own code: requiring production rather than the comprehension that they have been doing so far. In chapter 2, Brooks' work was discussed (Brooks, 1977), who suggests that writing programs consists of three main stages: understanding, planning and coding. Subjects experienced problems at all these stages, but the distinction that Brooks makes between planning and coding was not found: their planning involved producing bits of code, and when coding they would stop to re-think their solution. It should be remembered, however, that Brooks' categorisation was derived from a very experienced programmer writing code for a large program. There has been little work on how novices move between the different stages of programming.

Understanding the problem statement

In order to solve this problem and write the code they needed to note that the problem was of the form: If A, or else B AND C is present, DO. Four of the subjects in group 1 misinterpreted the problem and tried to solve a different problem: e.g. one of them interpreted "define your own procedure" to mean that she could define the problem:

1 What I'm trying to do is only if all three conditions exist is to deem him unhealthy.....

2 Aha...I see! I'm trying to work out my own definition whereas they're telling me what the definition is."

Protocol extract 6.6: subject working on ASSESS problem

Another subject thought that the information "whisky is unhealthy" etc needed to be put into the data base:

1 How do you do the unhealthy bit? It's not going to know that anything is unhealthy is it unless you've given it to it in a sentence.....

3 ...So do we need to tell it, to start with, whisky is unhealthy?

[E] No, you don't need that because you can have a simple print statement.

Protocol extract 6.7: subject working on ASSESS problem

Her interpretation of the problem is harder, in that it involves an inference (whisky is unhealthy, therefore if someone drinks whisky they must be unhealthy). Given that the context of using SOLO is cognitive psychology, and it is used as a modelling tool, this interpretation is not unreasonable. Another subject who misinterpreted the problem statement was from the second group. Extracts from his protocol are given below, and commented on in some detail. He is attempting to solve a problem analogous to strictassess.

2 .....Then you'd have to, you're defining your own procedure called ASSESS so I suppose it's the same kind of thing as the strictassess where you're er

3 Step 1 ... check if they did or did not drink whisky and if they didn't, then depending on the way you did it, you could.. have it exiting there, and, ... hold on, the ..printout is supposed to be unhealthy so if they did drink whisky you'd get it continuing onto the next check which is do they smoke cigarettes and then if they did.. you'd merely get in on the next check for if they drink beer as well, and if they drink beer you'd continue to line 4 which would be unhealthy.

[E] OK it might be helpful to actually work it onto paper as well.

6 ....1 would be check X drinks whisky, 10A if present, ... continue, 10B if absent print healthy exit

7 Uh, and so if they basically smoke cigarettes and drink whisky I'm going to say that they're pretty unhealthy, so it will continue to there, print unhealthy

[E] You've got a beer one as well in there haven't you

8 A beer one as well, OK, then right well it isn't very.... I think I'm going to have to change this because some people could, say, just drink beer and just drink whisky and not smoke cigarettes and they could be judged fairly healthy. So maybe it has to be more finely tuned.

[E] If you look at the question, it's whisky on its own or beer combined with cigarettes.

10 .....So in that case, we would, if X drinks whisky, if present, you'd then print unhealthy and exit, and if the person doesn't drink whisky you just continue to find out further information about them. You check if X smokes cigarettes, if present continue, if absent. Oh hold on, well if absent they may not drink beer but it doesn't really matter because they're going to be healthy because the criteria for being unhealthy is to drink and smoke at the same time. So that could stay the same. 30A could be check X drinks beer, 30A if present continue.
So it's the fact that they smoke and drink. Check X smokes cigarettes, if present continue to find out if they also drink beer, if absent print unhealthy exit. If absent, so that's if they don't drink beer, but they smoke cigarettes, print healthy, and that's it I think.

Protocol extract 6.8: S13's protocol of the ASSESS problem

STRICTASSESS is mentioned (2) and so presumably drawn on as an analogy. However, it may be cued by name rather than function as there is no evidence that it is understood as a functional schema. He hasn't read the problem statement carefully. His outline procedure (see segment 3) is given in figure 6.7 below:

```
10 CHECK /X/ DRINKS WHISKY
    10A IF PRESENT: CONTINUE
    10B IF ABSENT: (EXIT)
20 CHECK /X/ SMOKES CIGARETTES
    20A IF PRESENT CONTINUE
    20B
30 CHECK /X/ DRINKS BEER
    30A IF PRESENT PRINT UNHEALTHY
```

Figure 6.7: S13's first solution to the ASSESS problem

The exit is bracketed at line 10B as he is unsure of this. He never specified 30A or 30B

but he is solving a different problem, one which is in fact the same as STRICTASSESS, which is, if someone drinks whisky, drinks beer and smokes cigarettes they're unhealthy. This is also an easier problem, as it is clear from other protocols that some students have difficulties achieving the 'If A or B and C' logic of the actual problem using SOLO. He begins to talk about taking the EXIT branch at line 10B (3) but then follows the CONTINUE path through - again we know that learners often forget the ELSE branch of an IF ELSE statement, which is why SOLO prompts with IF ABSENT. However, in the first attempts at mental execution the IF ABSENT branch is often ignored.

In segments 3, 4, 6 and 7 he continues with the STRICTASSESS model, adding 'beer' in when he's reminded. He comments (8) that he's not very happy with the STRICTASSESS model, and it would seem either that he has his own model of what ASSESS should do or has remembered what the question wants. When he is reminded of the question, he changes his outline to:

```
10 CHECK X DRINKS WHISKY
    10A IP PRINT "UNHEALTHY; EXIT
    10B IA CONTINUE
20 CHECK X SMOKES CIGARETTES
    20A IP CONTINUE
    20B ?
30 CHECK X DRINKS BEER
    30A CONTINUE
40 PRINT HEALTHY
```

Figure 6.8: S13's second solution to the ASSESS problem

This solution is another example of ignoring the ELSE branch. He leaves line 20b with a question mark (it should be EXIT) and omits line 30B (also EXIT) altogether. Apart from this, the procedure is correct.

Planning and coding

It is hard to distinguish a planning and coding stage in the subjects' behaviour, especially with a small problem. Most of the subjects were impatient to get to the code, and their planning consisted of talking though the SOLO code they might use, for example:

1 You'd start off with assess, and then you'd have to, um..., um

2 You'd have to put in if X drinks whisky then he's unhealthy..you'd have to write they would become unhealthy or X becomes unhealthy, if absent, that would be exit, wouldn't it."

Protocol extract 6.9: S8 working on ASSESS problem

Subjects were discouraged from plunging straight into coding, and asked to think about what the problem involved, and to write a solution in English before working in SOLO.

It is in the planning stage that learners often use other procedures they have encountered as analogies. The problems that arise here can be thought of as instructional in that they are concerned with the subjects' approach to learning: in this case to solving the ASSESS problem. Four subjects referred to other procedures when attempting to solve the ASSESS problem. Three of these subjects did not understand the examples well enough to use them, or do not understand the ways in which the program they're currently working on is similar or different to the program they know. In other words they attempt to use examples to abstract plan knowledge but often fail. In the protocol below S14 thinks of the ASSESS procedure (2) but doesn't mention it again. She refers only to ASSESS, not WEAKASSESS or STRICTASSESS, but does refer to unfit (1) which is part of both WEAKASSESS and STRICTASSESS before changing it to unhealthy.

1 Well in the first place you'd have to check if X drinks whisky and if present you'd have to print out unfit, uh oh, unhealthy, but there's no alternative for....so

2 I was just thinking of the same procedure.....as to assess, just to assess

3 And the first test is to print out healthy if this person drinks whisky..... But I'm not quite sure if you're supposed to combine all this in one thing, like in there you've got there X is a man, X

plays squash, X rides bicycles. I'm not quite sure if you're including that into one parameter - X drinks whisky, X smokes beer and X smokes cigarettes.

.....

(E) When you say one parameter, what do you mean?

5 Are you talking about different people or is that not really relevant?

(E) It's saying that whoever you applied that to, right, which could be anything that someone had put in with some information, that it would be able to apply that procedure to them, and that if they drank whisky it would say they were unhealthy or that if they smoked cigarettes and drank beer it would way they were unhealthy. Does that make sense?

6 Yes I think so, as far as I can see, the first thing is to check if X drinks whisky, then if present, print unhealthy if absent exit. Part 2 goes on to check if X smokes cigarettes, um if present continue, if absent.....

7 You see I'm a bit confused because you want it to continue to check if he also drank beer as well, but I'm not sure what that exit would mean, if he didn't actually, yeah that's right I think, because if it was present he would go on to, he'd continue to check if he drank beer, if it was absent, it would exit and then it would check if he drank beer, and if that was present you'd print unhealthy and if absent you'd exit.

(E) That sounds alright, yes, now try and write it down

7 I can't quite remember what I said there.....

Protocol extract 6.10: S14 working on the ASSESS problem

The subject here is unsure about how to represent data for this problem. It's not clear what her model of the problem is. The requirements of the problem are that there are triples in the database of the form "X smokes cigarettes, X drinks beer, X drinks whisky", whereas S14 seems to be considering triples with a different first node: "X smokes cigarettes, Y drinks beer, Z drinks whisky". Another characteristic illustrated by this protocol is her tenuous grasp of the solution: having just verbally described the program, she cannot remember it in order to write it down.

Few subjects showed evidence of having a plan for solving a problem, or even part of a problem. They are therefore casting around looking at examples in the text but often have little understanding of the function of such example programs, i.e. they don't know which SOLO constructs to use, to achieve, for example, a "filter" or "disjunction". This is shown clearly in the difference between the first two protocol extracts of the ASSESS problem and extract 6.11 below. All three subjects are from group 2 and they use procedures given as examples in the text as models, but only the third subject is completely successful, and this is because she is clear about what each of the examples in the text achieves: i.e. she has an accurate abstract functional model.

She had already solved the problem when the experimenter went in, and is explaining how she did it:

(E) Could you please explain what you did?

1 Well here we have to deem someone unhealthy if they either drink whisky or if they both smoke cigarettes and drink beer,

2 So what I did was to combine the weakassess and strictassess type programs here, so, here we've got line 10 if drinks whisky, print unhealthy and exit. If absent continue.

(E) Could you explain how you were using that as a model?

3 That's the weakassess model.

4 The next two takes part of the strictassess model, so.... somebody has to smoke cigarettes and drink beer to be deemed unhealthy, so therefore line 20, check somebody smokes cigarettes if absent print healthy

5 I didn't know if it was right to do healthy and unhealthy, - it doesn't specify that.

(E) No, it's asking you to find some way to indicate they're on that branch.

6 So, if present continue and it goes to line 30, check drinks beer, if present print unhealthy because by definition if it's got to drinks beer it's had to have been smoking cigarettes as well. If absent continue and then at line 40 print healthy

Protocol extract 6.11: S15's attempt at the ASSESS problem

At the coding stage one subject did not know how to use SOLO to achieve her goal: "I know what I want to do, but I can't see how to do either or". In fact, how to do this is illustrated by WEAKASSESS, but it is not described in the manual in this way (see fig 6.3), and so may not be accessible to someone who is searching for "how to do either/or".

Four of the subjects tried to use a CHECK statement within another CHECK statement, i.e:

```
CHECK /X/ DRINKS BEER
    IF PRESENT: CONTINUE
    CHECK /X/ SMOKES CIGARETTES
        IF PRESENT: PRINT"UNHEALTHY" etc
```

A check statement is not permitted as a sub-line of another check statement. However, it is closer to the way that the solution is expressed in English rather than SOLO.

Constructing interpretations

Some examples of learners' interpretations have already been given, misinterpretations of what the WEAKASSESS procedure does and also of the ASSESS program. Related research on text editing discussed in chapters 2 and 3 found that learners construct hypotheses to explain events, or the behaviour of the machine (Lewis and Mack, 1982). But they often do this on the basis of little evidence. Hypotheses are set up, and retained, even in the face of conflicting evidence. An example of this is seen in protocol extract 6.12. S13 is answering the question: "How would you define the new procedure called PRAISE?" which is set in the text as a self-assessment question: the question is asked and the answer given immediately so that the student can check their understanding. The experimenter covered up the answer for this study and asked the subjects to answer and talk through their answers before they uncovered it. The context of the question is the introduction of hierarchical structuring of programming. The JUDGE procedure is given in figure 6.9, and the text immediately preceding the question is in figure 6.10.

```
TO JUDGE
1    PRINT "HERE'S WHAT I THINK"
2    CHECK /X/ VOTES INDEPENDENT
          2A IF PRESENT: PRINT "HOORAY", CONTINUE
          2B IF ABSENT:  PRINT "BOO"; CONTINUE
3     PRINT "AND THAT'S ALL"
```

Figure 6.9: The JUDGE procedure

"Now instead of simply printing HOORAY at sub-step 2A, we had wanted to activate several different procedures, for example the following three:

```
PRINT: "I THINK"/X/ "IS QUITE REMARKABLE"
PRINT "I REALLY RESPECT" /X/
PRINT "HOORAY FOR" /X/
```

Since we are only permitted to activate one procedure at any given step, what we must do is decide what single overall thing we would want to do at sub-step 2A which would encompass all three of the above procedures - in other words, we must try to group those three procedures into one single new one.

Let's suppose that we decided that those three procedures involve (roughly) an act of 'praise', and so we decide to define a new procedure called PRAISE

How would you define this new procedure called PRAISE?"

Figure 6.10: Extract of text on hierarchical structuring of programming

4 Well it is possible that you try and activate it if line 20A is the relevant line so rather than print hooray you could, ...mmm...maybe it's not legal, I have a feeling it's not legal

5 I want to on 20A, ... If present I want to print a new procedure called praise.

6 ..Could I insert it into line 20A could I insert if present um....continue to line 30 delete and that's all and at line 30 type in something like to praise and the definition, no, that's not legal is it?

(E) ...What is it you're not sure about?

9 How to define praise. Where to actually write down the definition of praise.

(E) Have you got any ideas about where you might do that?

10 Well over and above print on line 20A a .. longer version of hooray I don't know. Is that what one would do?...How would one actually activate a new procedure within the procedure to judge?

12 We want to activate several different procedures instead of printing hooray at sub-step 2A

13 Would we, um well just sort of type out here separately a new procedure for praise? ...Oh well I'll just try it. (Defines a new procedure called praise)

16 ...Quite how I'm going to activate this praise procedure whilst in the middle of judge I don't know.

20 .....I don't feel there's been any indication that once SOLO's been looking through this judge procedure that it's then able to sort of shoot down to line 3 and then suddenly go off and do an investigation of the praise procedure and then go back to judge. There's been no indication of that.

27 ..I'm going to try ..sticking in praise in 20A, ..though there's been no indication that it's accurate.

30 .....OK judge Fred, so, it does work, so you can incorporate just another procedure just anyway,.....I thought it would require experimentation

31 I thought I'd read earlier that you couldn't. Since we are only permitted to activate one procedure at any given step

(E) One procedure at a given step

32 Even if you're within another procedure it doesn't matter, that's what you're saying?

Protocol extract 6.12: S13 is answering the question "How would you define praise?"

The protocol extract given in protocol extract 6.12 clearly shows the problems this subject has in doing this exercise, and how he constantly refers back to the text to check out his model. From the beginning he can only see one possible real solution (4, 6), other than having print statements: "a much longer version of hooray" (10), and this is the correct one. He does not pursue this solution, however, as he believes it cannot be the solution required as it would be doing more than "one single overall thing". He is

also confused by where the definition of PRAISE will be (6, 9) and where the procedure will be activated (16) which contributes to the problems he is having. Finally he defines a PRAISE procedure (14) and also tries out the old JUDGE procedure (16), - but still doesn't see how it can be activated from within the JUDGE procedure. The core of the problem is contained in (20): he realises that if the only solution he can imagine were to work, control would pass to the new procedure and back to JUDGE. This is his correct model of the solution. At the same time he believes that this solution is illegal - the text has led him to believe that this could not happen, and so he is unable to carry out the correct solution because it conflicts with his inaccurate mental model of what is required. He believes that to execute praise would be to have more than "one overall thing" happen at sub-step 2A, because there are, therefore, two procedures, JUDGE and PRAISE. He is surprised (31) when this solution works, and finally realises his misinterpretation (32)

Like many other novices, when he was unclear, he read and re-read the text, scrutinising it and comparing it with his own hypothesis about how the praise procedure must work. This belief that you can't have two procedures is so strong that it prevents him from trying out the solution he thought of right at the beginning. Again this is an instructional problem.

Learning by analogy

Examples have been given of learners using example programs in the texts as analogies. There are two main different kinds of use of analogy here: one is the use of programs encountered in the text as models. As we saw in the ASSESS problem, success will depend on understanding the program in the text so that it can be used in solving the new problem. The text example then becomes the base domain (see chapter 3). Protocol extract 6.11 illustrates the successful use of this strategy, and the relevant parts are reproduced below. Here, the subject had already solved the problem when the experimenter went in, and is explaining how she did it:

2 So what I did was to combine the weakassess and strictassess type programs here, so, here we've got line 10 if drinks whisky, print unhealthy and exit. If absent continue.

(E) Could you explain how you were using that as a model?

3 That's the weakassess model.

4 The next two takes part of the strictassess model, so.... somebody has to smoke cigarettes and drink beer to be deemed unhealthy, so therefore line 20, check somebody smokes cigarettes if absent print healthy

5 I didn't know if it was right to do healthy and unhealthy, - it doesn't specify that.

(E) No, it's asking you to find some way to indicate they're on that branch.

6 So, if present continue and it goes to line 30, check drinks beer, if present print unhealthy because by definition if it's got to drinks beer it's had to have been smoking cigarettes as well. If absent continue and then at line 40 print healthy

Source: protocol extract 6.11: a successful attempt at ASSESS problem

Here the problem is clearly understood, and the use of WEAKASSESS and STRICTASSESS as models is explicitly stated. Unlike the earlier protocols of this problem given in protocol extracts 6.8 and 6.10, this one indicates access to a plan for this problem, directly derived from the examples. It is clearly understood that the weakassess checks for a match of triples, and exits as soon as one match is found, whereas the strictassess model is a search for a conjunction of conditions, so that for this to be achieved in SOLO, control must go to the next check statement.

A different kind of analogy is that drawn from other domains. Consider the following short protocol extract given below.

To Praise /X/
I think of it in relation to a word processor: that if I were doing a lot of letters I would do a letter and put an X in, Dear X, and each one I'd just put in Fred, Mary.

Protocol extract 6.13: Commenting on how a procedure takes a parameter

In this case the student was being introduced to the idea of a procedure taking a parameter. She was commenting on the title line, "TO PRAISE /X/". The word processor analogy is very useful, and appropriate in this context; which is the notion of

a procedure taking a parameter. The learner is able to make sense of a new concept through its connection with existing knowledge. This subject had never used a word processor; nevertheless she held beliefs about the ways they behaved, which seemed to be, for the most part, useful and accurate. The office she worked in, however, was about to acquire one, and she was going to be partly responsible for staff training and so it was a very relevant issue. Also, this is a spontaneous metaphor in that it is not given or suggested in the manual. However, although it is a good metaphor in terms of the parameter attribute, for the subject who produced the above protocol it led to expectations about how the editor would behave, - that it would be able to delete single words within a line of the procedure, which did not match its actual behaviour.

Here is a second example. The two procedures TO GRUMBLE and TO FOO, given in figure 6.11 below are presented in the manual, as the first examples of how a procedure can "summarise" many tasks, - in this case to PRINT three statements or to NOTE them, i.e. add them to the data base:

```
TO GRUMBLE                              TO FOO
PRINT "I DON'T FEEL SO GOOD TODAY"      NOTE BAZ GLUB FOZ
PRINT "MY FEET ARE SORE"                NOTE BAZ BIF ZAP
PRINT "BESIDES THAT, I HAVE A HEADACHE"
```

Figure 6.11: Two example procedures taken from the SOLO manual

The verbal protocols of one subject suggest that she is seeing the procedure literally as a verb. An extract of her protocol is given in extract 6.16 below, and indicates a confusion which the experimenter does not alleviate by continuing the verb metaphor, as he did not realise what was causing the confusion until the end when it becomes apparent that the verb metaphor is constraining her understanding quite strongly. A procedure can be usefully thought of as a verb, in some ways; however this subject believed that as such it behaved like a *particular* verb, e.g. grumble, and it has to have one particular format, and this was the SOLO primitive that it contained, NOTE or PRINT, so that it did one particular thing, NOTE or PRINT or whatever. Unlike a

procedure then, a verb is seen to be doing a particular job: to fit the metaphor, the notion of a procedure is constrained so that it is no longer a general function which can carry out a variety of tasks. This subject didn't have the appropriate mapping of verb to procedure.

1 I don't understand this; it says to grumble here. To grumble, to foo, they're both still verbs, right? I don't understand why that isn't the same format as that, I mean how can you apply those to a verb?

[E] What you're doing is creating a verb, as it were. You're telling it what to do when you use this verb in the future. This is just the name, to grumble, and it becomes like a little package.

2 But you could put any verb in there at all, couldn't you, like to jumper something....? But it's still got to be applicable to that verb, but what are you telling it here, that's still a verb?

[E]...these are just symbols...'to foo' is meaningless and so are all these things. But the machine will faithfully do it all.

3 But why has it got note here and print here? I just thought if you wanted it to do something it was a particular format. I see what you mean in a way, but I don't see why it's got print here and note there, surely, if it's got one particular format applying to verbs it's got note for one and print for another?

Protocol extract 6.14: Seeing procedures as verbs

6.7 DISCUSSION

It was pointed out earlier that most of the data discussed in this chapter is qualitative, and focuses on the kinds of difficulties that subjects had rather than how many subjects had them. The later chapters on DESMOND and Logo contain more quantitative data. It is often difficult to quantify the incidence of a particular problem: that four subjects mention a particular problem does not mean that the problem has not been experienced by other subjects - only that it does not appear in their protocols. Where possible the incidence of particular difficulties has been given, but this may be conservative. In any case, with 13 subjects, too much weight should not be attached to the proportion of subjects experiencing particular problems. The emphasis is on the problems themselves.

The subjects had two main kinds of difficulties: difficulties with the domain itself, i.e.

learning SOLO, and difficulties in how they approach learning SOLO and the problems set in the instruction manual.

Although SOLO's flow of control is made more explicit than it is in some languages by forcing users to specify the ELSE branch of the conditional, most of the subjects (10/13) had difficulties in understanding the control statements and knowing when and how to use them appropriately. They resisted putting in the CONTINUE or EXIT statement and often only entered them when prompted by the computer. Such difficulties in using control structures is a well known programming problem (Green et al 1980b, Miller, 1975) and so this result is quite consistent with established findings. Part of the difficulty may be that some SOLO constructs are similar to English words, e.g. NOTE, CHECK, FORGET etc., and this may increase the tendency towards anthropormophism. The data suggests that at least half of the subjects (6/13) attributed the procedures with some intelligence.

The belief that a program can have access to knowledge additional to that given in the program is described by Pea in a paper on language independent bugs. (Pea, 1986). He describes three different kinds of bugs which he suggests all derive from a superbug which is the idea that:

"there is a hidden mind somewhere in the programming language that has intelligent interpretive powers. It knows what has happened or will happen in lines of the program other than the line being executed: it can benevolently go beyond the information given to help the student achieve her goals in writing the program"

An example of such a belief was seen in protocol extract 6.3.

These two problems: the difficulties of using control statements competently, and everyday beliefs and knowledge conflicting with the programming knowledge that they were acquiring led to problems in writing their own code. Four of the subjects used procedures given in the text as analogs, but only one understood them sufficiently

to use them successfully. This suggests that novices are not able to abstract plans from the examples they are given in their instruction manual, although the most successful do. The result is also consistent with the fact that SOLO's conceptual model provides state descriptions and procedural descriptions but little in the way of functional descriptions at the appropriate level for learning the plans that they need. It cannot be presumed, therefore, that novices have learnt such plans, or can abstract them easily, and so it is necessary to provide plenty of examples of similar problems and be explicit about the ways in which the problems are similar: i.e. that they are isomorphic and use the same plan. This would result in a more "balanced" conceptual model which would provide a functional description in addition to the procedural and state descriptions. The various stages of solving a programming problem could also be made explicit and taught.

The subjects' weak knowledge of SOLO's control structures is rather like the fragile knowledge identified by Perkins and Martins (see chapter 2) who discuss students who are not able to "marshall enough knowledge with sufficient precision to carry a problem through to a clean solution." In this study, this is exemplified by S14 who produced a solution for the ASSESS problem, but then was unable to remember it in order to write it down, and the subjects who confused the EXIT and CONTINUE statements when tracing the WEAKASSESS and STRONGASSESS procedures. The interpretations that subjects developed were based on their own everyday knowledge and cues from the text.

It is not surprising that in a distance learning situation, learners scrutinize the text to test out these models or hypotheses: however it seems that such hypotheses are often sufficiently strong that they manage to interpret whatever they find as confirming evidence. In the case of SOLO subjects, the material they are learning is well structured, and very well taught, so they cannot be said to be literally "striking out into the unknown": which is how Carroll and Mack (1982) describe the behaviour of

people learning to use word processors. However, examples have been given where they do make their own sense of the material that they are encountering by setting up hypotheses to explain events which occur.

Lewis and Mack (1982) describe the explanations generated by learners as abductive reasoning, where a hypothesis is generated to account for one or more observation. Although such abductions are wrong, they suggest that abductive reasoning "has value in interpreting future events where there is no alternative to abductive reasoning" as complex learning is characterised by incomplete and ambiguous information. Whilst it is true that learning SOLO is complex, the quality of the instruction is extremely high, and as we saw in chapter 4, the conceptual model is well designed to meet the needs of novices. It is somewhat surprising, then, to find similar problems in SOLO to those found in other studies where no conceptual model was provided and the instructional material was not designed by a team experienced in distance teaching.

There are two probable reasons why some problems remain. In order to learn, it is necessary for the novice to interpret new material in terms of the knowledge she already has: the theories about how this happens in complex learning situations such as text editing or programming suggest that forming hypotheses to explain events is an important element of this process. So even if the text is well structured and taught, it is likely that it is necessary or important for learners to take an active, questioning, role, in order to make their own sense of the information. In doing so, they are inevitably going to make incorrect hypotheses at various points in the process. Secondly, they are learning from self study materials, without any face to face guidance. In this respect, their situation is similar to those of the learners in the other studies mentioned. In such a situation, when a learner develops a mental model which is incorrect, he or she can only resolve the dilemma by re-reading the material to check their understanding, and/or setting up experiments to try out their hunches. The subject whose protocol was given in protocol extract 6.12 did both of these, and successfully

resolved his problem.

Just under half of the subjects (6/13) used analogies in their learning, both from their everyday knowledge and from the text. It is harder to predict the analogies that subjects use spontaneously, and therefore a self instructional text cannot take account of them. They can be both helpful and misleading, depending on the analogy chosen and the mapping that the subject makes. The word processor analogy worked well for one subject learning about how procedures take parameters, but gave her false expectations about the SOLO editor which did not work like a word processor. The subject who developed the "procedure is a verb" analogy, developed an insufficiently general model of procedures.

It is likely therefore that learners will attempt to use problems given in the text as analogies. Unfortunately if, as is often the case, they do not have functional plan knowledge of SOLO, they do not understand the problems sufficiently to use them successfully. They are also unlikely to make an appropriate mapping. Conway and Kahney (1987) show that learners can be helped to use the problems given in the text by pointing out the mapping and giving them functional information. (In this study it was a definition of recursion).

6.8 CONCLUSIONS

Subjects did have problems in learning SOLO, both in understanding programs that they encountered in the text and in writing programs. Problems centered around the understanding and use of control statements, inference limitations, understanding problem statements and constructing interpretations.

The study focussed on students learning in a "natural" setting and so to a large extent, the problems encountered were constrained by the curriculum, and it was not possible

to quantify many of the problems. It should be noted, however, that a certain set of problems were absent. These are low level syntactical problems or problems understanding the error messages. Students were encountereing interesting problems rather than being stuck on trivial problems.

About half of the students learning SOLO go on to complete a relatively complex project at summer school and more would do so if there were more places available. In this sense, bearing in mind that many students start learning SOLO with no programming knowledge and sometimes with a less than positive attitude, it is certainly successful. This chapter has discussed the difficulties that students encounter in their early days. Whilst many students recover from these and go on, for some students the frustration and sense of failure deters them from future programming efforts. For this reason, it is important to investigate and understand such difficulties.

Chapter 7

PT501

Contents

7.1 INTRODUCTION

This chapter describes the difficulties encountered by learners of an assembly language. Low level languages have potential benefits for novices learning via a conceptual model as they are closer to the machine than high level languages (which can be closer to the problem). It is easier to teach students concepts such as "variable" in a way that helps them develop an accurate model of the machine, because they can trace the movement of data from register to register. The notional machine can be made transparent more easily. The argument for learning both a high and low level language is that it gives the novice a view of the machine at two quite different levels: a global or macro view and a micro view, which together should help the student form an accurate model. Previous research in this area has been inconclusive. Weyer and Cannarra (1975) reported that children were able to learn Simper and Logo nearly simultaneously, and did so faster than students who learn the same languages sequentially, but for the reasons discussed in chapter 2, this claim is rather weak.

One aim of this study, therefore, was to investigate the effects of learning a high level language on subsequently learning a low level language and vice versa. The languages chosen were SOLO and an assembler used as part of a short Open University course (PT501), for the reasons discussed in chapter five. Two studies were carried out. Due to the high attrition rate on PT501, and instruction-related problems, it wasn't possible to draw any conclusions about the effects of learning a language from the first study, but this study did reveal some interesting findings which were followed up in a further small scale study. For this second study a group of three subjects were recruited, of whom two completed the work. These are the same three subjects who completed the SOLO work reported in chapter 6. This chapter will therefore report on the problems experienced by the subjects in learning the PT501 assembler, drawing on data from each of the studies in turn.

7.2 STUDY 1

There were two aims of this study which were:

1 to investigate the effects of learning a high level language (SOLO) on subsequently learning a low level language, PT501 and vice versa;

2 to investigate the kind of problems experienced by subjects learning to use the PT501 microcomputer.

Half of the subjects did not complete the work on the PT501 microcomputer, and had various difficulties with it and for this reason it was not possible to achieve the first aim and investigate the effects of transfer. The analysis of results will therefore report on the second aim: the difficulties that subjects had in learning PT501.

Subjects

For this study, cognitive psychology students who would be studying SOLO as part of their course, were recruited. After some initial drop out 16 home experiment kits were sent out to students on the Open University Cognitive Psychology course who volunteered to take part.

Procedure

The subjects were sent their kits a short while before they began studying the SOLO manual as part of their work for the Cognitive Psychology course. Half of the subjects were asked to work through the PT501 book before they worked through SOLO, and half were asked to wait until they had finished working through the SOLO manual. They were asked to work through the experiments in the booklet, answer all the in-text questions and write their answers on the experiment booklet. Subjects also

commented on the booklet itself. They worked on the experiment book at home and in their own time. They were told that they should allow at least 10 - 12 hours for the work, but that they should work at their own pace, and that therefore the time taken to complete the work could be expected to vary considerably from subject to subject.

Data collected

The kits consisted of the experiment book for the course, (which was commented on by the subjects) and the microcomputer. The subjects filled in an initial questionnaire on their attitudes to the role of AI in the course, to the use of computers more generally, and stating any experience of computers that they had, although all had been selected as 'novices'. Final questionnaires were sent to the subjects two months after they had finished working through the experiment book. An example of filled in initial questionnaires are included in appendix 7.1. Concept interviews were tape recorded at the university, and consisted of asking the subjects to describe, in their own words, what each of the concepts meant to them. The concepts are given in appendix 7.2 as part of the final questionnaire.

Most of the subjects came to the university to work on SOLO in the laboratory. (It was intended that all the subjects should do this, but for various reasons it was not possible).

7.3 RESULTS

Experiment books

The subjects divide into two main groups: those who completed all the work and who apparently developed quite a good understanding of the microcomputer, and those who did a certain amount but gave up - and although they may have understood some of the work, were expressing a lot of confusion. There is some middle ground, but not much. Eight of the subjects completed all the work, and 6 of these had at least a

reasonable grasp of it, whilst two were rather unsure. Six subjects partly completed the work: of these, 2 did most of it, and these can be seen as forming the middle group: having completed most of the work, but not having developed a full understanding. Four subjects did very little and 2 did not attempt it at all. This group cited personal circumstances as the main reason for not finishing: illness, relatives' illness, etc. However, they also experienced difficulties in getting as far as they did, and some, like the middle group, found it very tedious. The time taken by the successful 6 was as follows:

1 10 hours

2 13 hours, 15 minutes

3 4 hours

4 No information

5 12 hours

6 8 hours, 40 minutes

The average time taken is therefore nine and a half hours, but with a large range: there is a difference of over nine hours between the fastest and slowest, which is nearly the average time taken.

Of these 6 subjects, 3 wrote spontaneously to say that they had enjoyed using the microcomputer: one referred to it as "compulsive"' another as "a very pleasant diversion", and the third as being "addicted". Another student in this group became very interested and wanted more information.

Although they worked at very different rates they all commented quite liberally on the text and annotated it - for example the state transition network on page 17 is annotated on two copies. In both cases the subjects had put a ring around "state 1 fig 4" and commented: "machine must be in state 1 to execute instructions", this annotation

presumably being made for emphasis to help them remember what was important. Three different people also noted that "address pointer is used as program counter", and it is possible that this comment is made because the dual use of the address pointer as program counter is a source of confusion. One person reported experiment 5 as very difficult.

The other two subjects who completed the work showed less evidence of understanding: one of these subjects commented:

> "Mostly I wasn't sure how many times to press E before SS and also on some experiments I had difficulty in getting the instructions to come up on the display lights. It was some time before I understood address and register and I still don't know what they mean. When I realized what the relationship between the instructions and the code was it made it much easier. A summary of the different moves at the end of an experiment would be helpful. Step 3 on experiment 5 was very helpful. However, I enjoyed using the microcomputer a lot and felt I understood it at the end."

It was noted in chapter 4 that the state transition network, which was intended to clarify the key presses needed to move from one state to another, was not at all simple. The other student who completed the book took 19 hours - the longest time - and said:

> "I found it rather difficult. I often had to go back and refresh my memory. My husband (non O.U.) joined me and found the language incomprehensible. We took 3 hours to do experiments 1 and 2. All the 1s and 0s got confused. The denary mode was easier as it makes sense to see on one side a register number and on the other side the contents. I could do the experiments with the temperature lights because there were step by step instructions. If time permitted I would go through again to get a better understanding."

One student did all except section 7, but seems to have had problems in places, especially experiment 5, (the instruction mode) where he finally gave up, having failed to get the expected response, and in experiment 7 (sub-routines) where the text states: "Has the addition been performed as you would expect?", and he comments:

> "I didn't know what to expect".

Other subjects had various reasons for not finishing, but at least three of them didn't find it at all engaging: "I couldn't relate to it"; "I lost all interest", and "it was boring", so this may be a significant factor for this group. One student got as far as page 61,

and although she had found it boring, she also said it was hard at the beginning and got easier as the process became more automatic. She felt that it gave a "wide picture of computers, was totally different to SOLO, but widened my knowledge, built on what I'd learnt". Experiment 5, the instruction mode, proved particularly difficult for her too, and she has annotated that "address pointer = program counter". Another student who had said it was boring also annotated the figures, had problems with section 3 step 7 and gave up at the end of experiment 5 (p21). Others gave up during the same section: section 3, the program memory. The difficulties experienced here, then, can be summarised as:

1 difficulties remembering the sequence of keypresses;

2 terminology;

3 difficulty relating the contents of locations to addresses (except in denary mode);

4 particular trouble spots, e.g. learning about the instruction mode;

Why did six subjects succeed in completing the book and not the others? It is not possible to answer this question definitively from the data collected, but there are indications of the likely reasons. There is no evidence that they experienced significantly fewer confusions than the others but it is likely that they took a more active problem solving hypothesis-testing role and were 1) thus able to resolve contradictions that arose and 2) remained more involved because they were active, and thus finished the work, actively learnt from it, and reported it to be enjoyable. The evidence for this is the amount of writing and annotations on their texts, which were written on and commented on much more than the others.

Summary of questionnaire responses

Subjects completed these questionnaires two months after they had finished working through the experiment book.

Explanation of terms

They were asked to explain various terms as though they were explaining them to someone who knew nothing about the microcomputer. The terms asked about were the following:

Accumulator, single step, calls, address, register, JIFO key, address pointer, increment key, program counter, data memory, data, program memory, RAM, ROM, program, subroutine.

The concepts everyone understood were:

Program, ROM, subroutine.

The most confusing were:

Calls, register, program counter.

Examples of answers marked as correct are:

Program:

The steps/operations in the exact order to be performed to make the microcomputer carry out its instructions

Set of instructions to the microcomputer

The set of instructions the computer user gives to the microcomputer to manipulate data to achieve certain ends or perform a task

Subroutine

Self-contained set of instructions forming part of the program e.g. JIFO

Part of the program in which a particular sequence is used more than once in the program. Writing those parts as a subroutine saves writing the instructions several times and saves on memory locations

Examples of answers marked as incorrect are:

Calls	The process by which the number on the display address has been typed out
	No idea. A key which when pressed displays the contents of an address?
	Exposition for 'calling up' a particular step or figure
Register	A prefix to the address I think, but I don't know why. Right hand group of digits perhaps?
	The address displayed visually
	The mode in which the computer is used (This is correct, but insufficient)
Program counter	During the program the address display will alter as the program is completed

Grouping the concepts

Subjects were asked to draw links between the concepts they perceived as related, and to explain how they were related. There were only two main different ways of grouping the data.

1. Where one element in a category includes or effects others in some way, e.g. program, program memory: the program memory stores the program. This is a kind of functional category which differs from elements belonging to the same class, e.g. data, data memory - data memory allows data to be stored; increment, accumulator, the inc key increases display of accumulator. There were 8 instances of this.

2. The second was to group elements in one category according to their function, or type, or a kind of miscellaneous bag. However, what subjects saw as belonging

together in a category varied considerably: e.g. RAM, ROM, JIFO were all classed as conditional instructions (but note that they're not!); RAM and ROM as both program types; calls, register, address pointer and address as being to do with access to data; program, data were in category of software, and RAM, ROM, program memory, data memory were in the category of hardware. There is little in common between the various categorisations, and many are based on incorrect understanding.

Relationship between SOLO and PT501

As was explained earlier, it was not possible to pursue the first aim of investigating transfer because of the high attrition rate and because of the instruction related problems that the PT501 subjects reported. However some subjects did comment on the relationship between SOLO and PT501. Few of them perceived much similarity between SOLO and PT501, although they understood that there were similarities at the level of the basic machine. Three of the subjects expressed the similarities as follows:

"on reflection the micro has given me a better understanding of what's going on, of how the various processes could be broken down into single operations, - to think in terms of procedure-directed as opposed to goal-directed behaviour"

"the basics are very similar, - the purpose of program data, subroutines etc; only details differ since usage differs because it was using more basic input data, gave insight into the workings of the microcomputer which one could safely overlook with SOLO"

"The programs were similar, i.e. both have regular steps, with subroutines calling up the program. Both used stored information from the data base"

7.4 CONCLUSIONS OF STUDY 1

The subjects fell mainly into two groups: they either did nearly all of the work or very little. There is various evidence that it is the early stages that are most difficult, especially section 3, program memory, which includes the experiment on executing instructions. Other difficulties include the address pointer and program counter and

the use of binary notation.

The subjects who gave up stopped at the same place, the section on the program memory, which suggests that the beginning, especially this section, is hard, and perhaps daunting, but if people get past the hurdles, they tend to stay the course. There was a difference in perceived interest between those who dropped out and kept going, in that those who gave up said it was boring and those who kept going found it interesting. The successful subjects tended to take a much more active role in their learning, they engaged themselves in the task more. They had the same problems as the others, but were willing and able to confront the confusions, and to test out hypotheses about their conflicting models. This meant that they were often (but not always) able to resolve the conflicts and also resulted in increased motivation.

At least some of the experiments, ironically perhaps mostly those nearer the end, can be worked through in a cook book like manner without necessarily understanding what's going on. Subjects can work through them without any particular goal. They then feel as though they're twiddling bits: things sometimes work, but they don't know how, and they don't know why errors happen (even when they're deliberately made following instructions) or why they're doing what they're doing. This feeling of being able to successfully carry out procedures, and sometimes get the right result, but without having understood the process has been expressed by Open University students in other studies. For example Kirkup's study of the use of home computing in an introductory computing course (Kirkup 1989) reports a similar finding.

Finally, regarding SOLO, two of the successful subjects did express the opinion (spontaneously) that it had helped:

> "Re D303..... using the micro has increased my understanding of the underlying processes involved in Artificial Intelligence."

In terms of the second aim, then, which was to investigate problems that subjects might

have; there were significant problems for subjects working through the experiment book at home, and a high attrition rate. This meant that it was not possible to investigate the first aim, which was to examine transfer, because there was not sufficient competence. Some of the problems have been discussed, but although some subjects commented quite extensively on their problems by annotating the book, it was not possible to observe the problems as they were happening, and so a lot of information was lost. This factor, plus the lack of motivation experienced by about half of the subjects led to a decision to observe a small number of subjects working through the book in the laboratory. This would enable close observation of problems at the time they were happening, and also the experimenter would be able to push the subject along when motivation was flagging. This study is described next.

7.5 STUDY TWO

Introduction

The first study raised some interesting questions, although as far as transfer was concerned, there was insufficient data to investigate this issue, because of attrition. Transfer depends on learners achieving a reasonable level of competence, and partly due to attrition, the subjects in the first study never reached this level of competence. The second study therefore aimed to investigate the areas of difficulty which had led the subjects to give up. It was suggested that:

1 the beginning of the experiment book, especially experiment 3, was a source of difficulty;

2 subjects were unable to develop accurate mental models of the PT501 machine, but they did develop misleading mental models.

Subjects

Three paid subjects were recruited to work through PT501 and SOLO. These are the

subjects referred to as group two in chapter six. One spent all his time working on SOLO and never began the work on PT501. Therefore this is a case study of two subjects. Up to two days were allocated for the work; one subject spent a day and a half on it while the other spent two days.

Task and procedure

Subjects were asked to think aloud as they worked through the exercises provided in the instruction booklet, and to describe what they were thinking as they tried to solve the problems set in the booklet. In the PT501 manual it is intended that students write their answers in the exercise book and the subjects did this. All the subjects' verbalisations were tape recorded. Initially the experimenter remained in the room with the subjects, to prompt them to think aloud, and then sat in an adjacent room to be "on call" should the subject feel totally stuck. The experimenter only helped the subjects solve the problem they were working on if they were unable to move on without help. Otherwise the subjects were prompted to explain their problem, but not helped to solve it, although as time progressed they might be pointed in the right direction.

Results

This section examines and discusses the protocol data of novices working and carrying out experiments on the microcomputer. Protocols have been deliberately selected to illustrate sections where subjects had some difficulties, and although the manifestation of the problems are different, the examples are all indications of the same problem: that under scrutiny the text misleads the subjects, and they misinterpret what is happening. This is illustrated with examples from three different areas. The first example is from experiment 3: "selecting a location in program memory and putting an instruction in it"; the second example is from experiment 16, "the LOAD and EXCLUSIVE-OR instructions", and the final example is experiment 17 (which follows straight on from 16):"the JUMP and JUMP IF ZERO instructions.

The protocols

In the protocol extracts that follow, the instructions from the manual are given in the left hand column. and the style of the manual (e.g. italics and bold) has also been followed. These are numbered according to the steps, as given in the manual. The subjects' comments whilst reading and carrying out the step are given in the right had column. Where the subject has gone back and repeated or corrected entries, (i.e. where there are more entries in the right hand column than in the left) the protocol in the right had column has also been divided into numbered segments (which are not related to the instruction steps) for reference purposes. In the discussion of the protocol, the segment or step referred to is bracketed. All the protocol extracts for PT501 will follow this convention.

Example 1: Experiment 3 Selecting a location in program memory and putting an instruction in it

The learner has been told that the micro has two types of memory: data and program, and in section 2 and experiment 2 she has looked at the data memory and has learnt how to select a register with a certain address and to examine or change the contents. When the new contents have been written into the relevant register the text says "notice that the display lights again show the current value of the address pointer (0000 0000) - to work with the data memory again you will need to press the register key, Reg." The issue of how the address pointer is used is relevant to experiment 3, where the student has to select a location in the program memory and put an instruction into it, as will be seen in the examples that follow.

The protocol extract given below, is of S14 working through experiment 3. She was clearly confused about what to expect and what should have happened. There are four different types of problem, here, which are related. First of all there are expectations of what will appear on the display. Secondly there is not enough information to check,

i.e. there is a lack of feedback. Thirdly the whole issue is conceptually complex, and finally, despite the notion of the address mode light signalling when an address is shown on the display thus making the address/content distinction clear (and visible), this signal doesn't work very well because of the way that the address pointer is used, and therefore the distinction is blurred.

Experiment book text	**Verbal Protocol**
1. Prepare your microcomputer for use	I've prepared the microcomputer for use,...um..
	...
**The 0s that are shown are the current value of the address pointer, which is 0000 0000. Notice that the address mode light is still on. You are going to use the address pointer to enable you to select the location with address 0000 0010 and put an instruction in it.*	got the current value of the address pointer....yes that's OK.
You can examine or change the contents of the address pointer whenever the address mode light is on.	
2. Set up the new address pointer value, 0000 0010 by pressing the keys 0000 0010.	I'm now setting up the new address pointer value, which is [presses keys] 0000 0010
Notice that address mode light stays on	and the address mode light stays on as it should be.
3 Check that you have set up the address correctly (use the clear entry key if you have made a mistake so that you can set up the correct value) then press the enter key.	Now I've set up the address correctly
	and I'm now pressing the enter key (step 3) and the address mode light has gone out.
The value of the address pointer is not changed until you have pressed the enter key.	
**Notice that the address mode light has gone out. When you pressed the enter key the value of the address pointer changed to 0000 0010. The microcomputer then examined the new address pointer's value, read the contents of the location*	

Experiment book text

whose address was the same as the address pointer's value and copied them onto the display lights. What you are seeing therefore is the contents of the location with address 0000 0010. It is not the contents of the address pointer. The value of the address pointer is always interpreted as an address. Figure 3 illustrates this. Because the system is showing the contents on the display lights, the address mode lights has gone out.

Verbal Protocol

Now it says here that there should be...? Yeah, I seem to have gone *the* wrong somewhere, so I'll just go back and do that again.

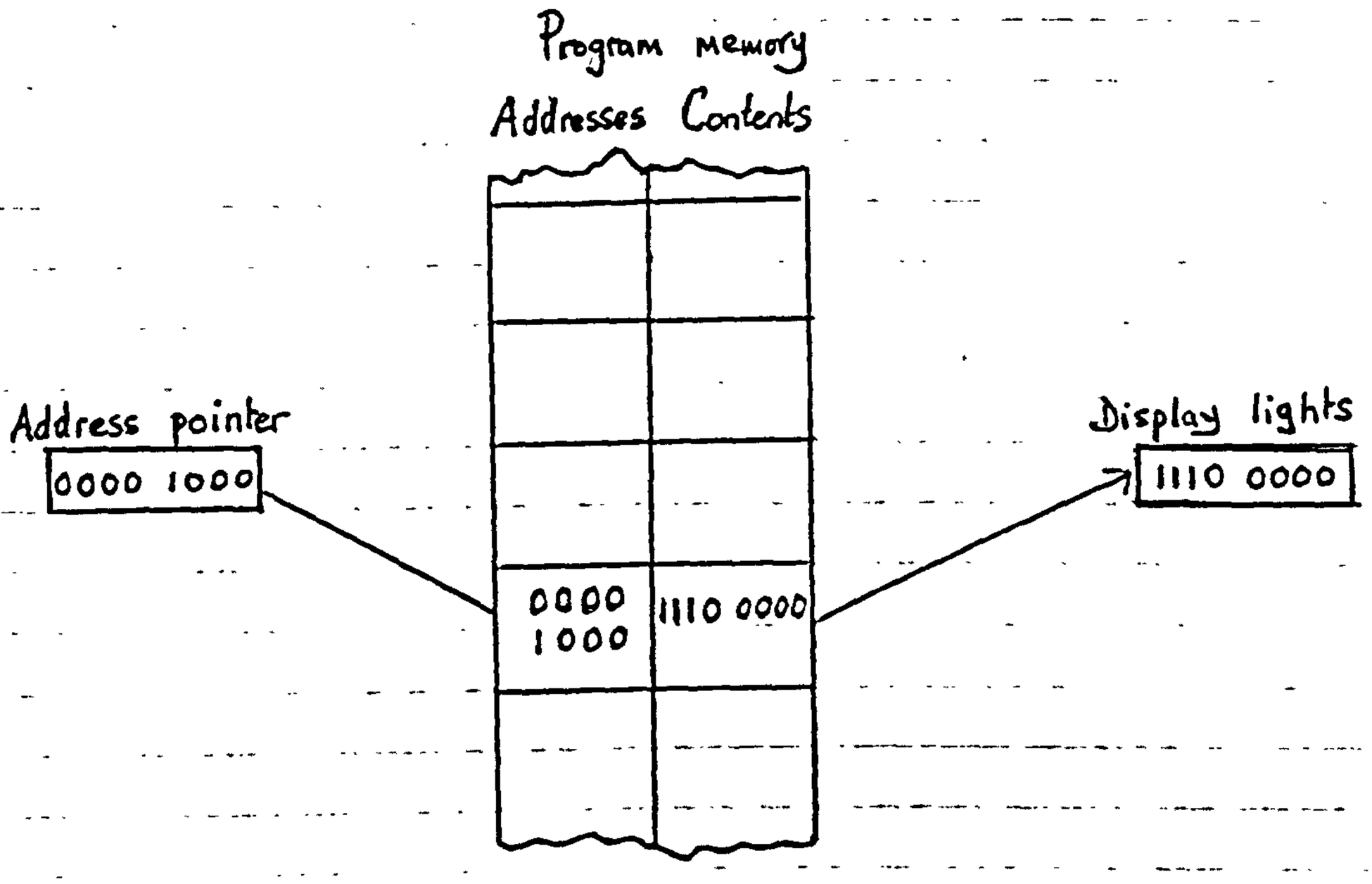


Figure 3 How the value of the address pointer is used as the address (0000 1000) of the location whose contents (1110 0000) are to be displayed (Source: PT501 experiment book)

Now I've just done that over again and I've got exactly the same result so I've obviously done something drastically wrong, but I'm not quite sure what.

Protocol extract 7.1: S14 working on experiment 3

Up to step 3, and pressing the enter key, everything is going well. At this point, the text reads: *"Notice that the address mode light has gone out"*, (which it has). This is where things start to go wrong. At this point her display shows the contents of the address 0000 0010, but this value is not given in the text. She believes that she should have 0000 0010 on her display, and this expectation that she has is confirmed by the continuation of her protocol below. However, on reading the next section of the text, it is explained that the display will not be showing 0000 0010 at this point: what happens is that the new value (0000 0010) of the address pointer is examined, and the contents of the location with that address are displayed; although the sentence beginning "The microcomputer is then examined....". is very hard to follow. If the subject can follow to the end, however, this point is made again. The value of the address pointer is always interpreted as an address. Figure 3 illustrates this, but shows different values for both the address pointer value and its contents to those in the experiment, so either it's wrong, or it's illustrating a different example. Maybe the values in figure 3 are different so as not to give the game away: i.e. not revealing the contents of the location with address 0000 0010: this seems the obvious explanation. But at this point the learner needs to know whether she is interpreting correctly. No answers to experiment 3 are given at the back of the booklet. S14's comment is:

> "You'd think that (what she had on the display) would occur somewhere wouldn't you, just within this bit to show that you're doing it right".

The continuation of the protocol is given in the next extract (experimenter comments are prefaced by square brackets):

Verbal protocol

[E] What have you done? What seems to be happening?

1 Well, I've carried out step 1, I prepared it for use, I put in that there, 0000 0010 and that's OK and then....yes, then I pressed enter and that's what I got. I've checked it three times and that's what it should be (refers to p16 of experiment book) which is quite different from what I've got here and I don't quite know why.

[E] Well how about going through it from the beginning as you did it to make sure

2 Right. I put that in there, pressed that to get that, then I put in the new address thingy as 0000 0010, that's what it's meant to be, and then I pressed the enter key, and

3 that's what it's supposed to be (refers to step 3) (Expects it to be 0000 0010?)

[E] (reads out explanation of step 3) in other words the address is now 0000 0010 and the location (means contents) is that..(1100 0000).....

Protocol extract 7.2: S14 working on experiment 3

It is clear that at step 3, S14 expects to have a different value on the display from what she has.

Her protocol is continued in the next extract below:

Experiment book text	Verbal Protocol
4 Write down the contents of the address 0000 0010.....	4 But there's no indication as to whether this is correct.
5 Now change the contents to 1101 0000.	5 I'm now changing the contents to 1101 0000.
You can do this in exactly the same way as	

Experiment book text

you would change the contents of a register: set up 1101 0000, check it and then press the the enter key

** 1101 0000 is the binary code for an instruction which can switch on selected lights in the traffic lights. You have now put this instruction into the location with the address 0000 0010.*

The contents of a location are not changed to what you have set up on the display lights until you press the enter key.

6 Notice that the address mode lights have come on again and the microcomputer is displaying the address pointer's value. Change this value to 0000 1010 by setting up 0000 1010, checking it and then pressing the enter key.

I shall refer to this process as *selecting a location with a given address*

Verbal Protocol

6 I've put that into the computer, and I've pressed the enter key

7 but the display has changed to 0000 0010 and here again I'm not sure whether that's meant to be on the display or not, so I'm a little bit unclear. It doesn't actually say anything's meant to change on the display -

8 yes, just going a bit further down it does mention that one so it must be right.

9 OK, so this figure must be the, .. this is the actual address of the location, is that it? No it's not, yes this is the pointer's value It now says change this value to 0000 1010 by setting up 0000 1010 so I'll do that and see what happens.

10 Right well that seems a bit confusing.

11 I think I'm going to go back to the beginning of experiment 3 and start again because it's not very clear.

12 Mm I've now gone through that again and I presume I've now got the location with the address number 0000 0010 (see step 5), the address mode light has come on again, and I suppose the computer system is displaying the address pointer value. Now the next bit is the bit where I went wrong I think.

13 Now where it says change the value to 0000 1010 by setting up 0000 1010, I've keyed

Experiment book text	Verbal Protocol
	that in and pressed the enter key and have come up with 0111 0011 which doesn't occur anywhere in the text, so it may well be a mistake although that may be the contents, yes it probably is.
7 Change the contents of this location to 0101 in the same way as you did for the location with address 0000 0010 in Step 5. (This is the binary code for the instruction which can perform an addition)	14 Now I've got to section 7 which is to change contents of the location to 0101 0001 and I'll just see what happens here; now I've keyed that in and pressed the enter key, I've got four noughts 1010. I'm now doing the instruction, no, I'm not.
8 Also put this instruction (code 0101 0001) into the locations with addresses 0000 1011 and 0000 1100.	15 Section 8 looks a bit confusing
9 Check that you have changed the contents of the following locations correctly	16 Well I've gone through 8 and 9 mmm. I've checked through the locations and written in the contents but as I say I'm not quite sure if these are correct, um, I don't think I quite understand the process yet so I'm going to have slight difficulty in summarising this as it says in section 10. I think I'll just go back and reread the instructions for this whole unit.

Protocol extract 7.3: S14 working on experiment 3

S14 was not able to interpret what appeared on her display at several different points (see protocol segments 7, 13, 14). At step 5 the display is showing the address pointer, but she doesn't know this (7), although she then reads the information in bold which explains it. Again (13) she doesn't know what the numbers on the display represent, but deduces (correctly) that it's the current contents, having remembered (12) that the address mode light indicates that the microcomputer is showing the address pointer's value. In 14 she changes the contents, and is again unsure of what the display represents. The display is once again showing the address pointer value, though S14 does not refer to the address mode light being on. Earlier, at step 3, the text inadvertently cued an expectation which was not correct: the

expectation that the display would contain a certain binary number. When the subject discovered that this was not the case she looked for help in debugging her model. Unfortunately help was not forthcoming because she wasn't told what should be on the display at this point.

There are four related problems.

1 Indirect addressing is a difficult concept; which is dealt with here, but the sentence which explains it is embedded and not easily understood: *"The microcomputer then examined the address pointer's new value, read the contents of the location whose address was the same as the address pointer's value and copied them onto the display lights"* (step 3).

2 The second problem is an instructional problem. There are three different type faces in this step, each of which conveys different information. The ordinary type face tells the user what to do, the bold type tells her what she should see (but in this case not clearly enough), and the italics offer a commentary. However, the role of the three different typefaces is not made explicit, and also the commentary contains information which is vital to understanding what's going on, but which can easily be overlooked as an additional interesting point to be noted. In fact this booklet was a prototype for later courses in which the three types of information were presented as three columns of text. Here the left hand column contains the instructions for what to do, the middle column shows the screen or display and the right hand column offers a commentary. The DESMOND booklet uses this system successfully and it is described in chapter 4.

The subject does not always know exactly when to expect the display to change, or to what, or what it means, see her protocol of step 5 where the description of the display comes too late, in bold type. Neither is the significance of the different typefaces clear to the learner. At instruction step 6 she clearly expects feedback about whether or not the display is correct at this point, but does not get it. Her final comment of "I don't think I

quite understand the process" is quite an understatement in the context of her protocol, yet she did work through the book quite successfully.

3 Although the address mode light signals when the address pointer is displayed, this subject is confused at times by whether the display is showing an address or contents.

4 One of the problems of the address/contents distinction is "the way the value of the address pointer is always seen as an address". This blurs the distinction and makes the conceptual model more complex.

Example 2 Experiment 16 The Load and EXCLUSIVE-OR instructions

A second example of the text "leading students astray" is shown in the protocol of the same student working through experiment 6. The extract starts at step 6.

Experiment book text	**Verbal Protocol**
6 Put the microcomputer into the binary mode and change the accumulator's contents to 0110 1111	I'm putting the computer in binary mode and changing the accumulator to 0110 1111 and I'm just checking it.
7 Put the microcomputer into the instruction mode and select the location whose address is 25.	
8 Start to put the instruction EXCLUSIVE-OR 1111 1111 into the location by pressing the key labelled Eo. Check that you have done so correctly and then press the enter key (not the increment key)	I'm starting to put the Exclusive-or - that's interesting, I've done that wrong. Cancel that, I'm just going back to section 7 and putting the micro into instruction mode; selecting location 25. I've entered that and am now pressing the Eo thing, what I forgot to do was to actually press

---

A two word instruction occupies two locations whose addresses are consecutive. An example of such an instruction would be JUMP 20 (which is the instruction referred to in the protocol extract). If the code for JUMP (which is 128) is put into the location whose address is 50, then the number 20 must be put into the location whose address is 51. When the microcomputer executes this instruction, it first reads the contents of the first location (in this case 50). These contents tell it both what the instruction is, and that there is a second word to the instruction. The program counter is therefore increased (to 51) so that the microcomputer can read the second word.

Experiment book text	Verbal Protocol
	enter after I selected 25. I've pressed the enter key.
9 Select the location with address 26	Change your location to 26, 26, enter
10 Change to the binary mode and change the contents location to 1111 1111 and go back to instruction mode	Change into binary mode and changing the contents of this location to 1111 1111
11 Check that you've done this correctly and then return to instruction mode	I'm checking that, oh that's interesting, it won't go back, I'll have to enter that, eh, that's interesting I'll just have to check that again, I think er, 26, I'll just have to check what 26 is in binary, right I've checked that, that's OK I've got it back to instruction mode I'm changing the location to 25, and I'm checking the contents which should be 96 which holds the code for the instruction.
12 Check that the location with address 25 holds the code for the Exclusive-OR instruction (96).	
13 Ensure that the program counter holds 25 and then press the single step.	Back to 25, single step, and they're showing 27
14 What do you expect the contents of the accumulator to be as a result of an EXCLUSIVE OR operation between 0110 0000 and 1111 1111?	Now, step 14, what do you expect ...(reads)..to be between that and that? I think I'm going to have to work them out, hold on. Right,well I think they should be 1001 0000, well I'll just check that, oh, um, now. I'm not quite sure, it looks as though I'll have to go back to binary mode. In denary it's 144 so I'll just check that, I've gone back to binary mode and it's OK.

Protocol extract 7.4: S14 working on experiment 16

In the above protocol there are indications of some confusions at steps 8 and 11. For this person, "that's interesting" always meant something like "what the hell's going on, that's not what I expected, I don't understand!" Another subject's protocol and filled-in experiment book clarified what the problem was: at steps 7 and 8, the words *"Select the location with address 25. Start to put the instruction EXCLUSIVE OR 1111 1111 into this location"* sets up the expectation that this will go into the location with address 25. In fact, as it's a two word

instruction, only the first part, the EXCLUSIVE OR will be going into the location with address 26. But supposing the learner's expectation, having forgotten that it's a two word instruction, is that the whole instruction is in this location. She would expect the contents of address 25 to be 1111 1111, or its binary equivalent, as it's the last thing entered. This is exactly what happened to this subject S15 and to S14 whose protocol extract appears above in extract 7.4. S15's protocol is not given, as it's somewhat confused, (and angry), but she deduces that step 8 is wrong, when she checks and finds 96 - the code for Eo. She finds the binary of 96, 0110 0000, and crosses out the 1111 1111 in the text, and replaces it with the 0110 0000, saying "how could they have made such a mistake?", so step 8 now becomes the instruction EXCLUSIVE OR 0110 0000 (0110 0000 = binary of 96, 96 is code for Eo), and of course compounds the problem and gets in a muddle. The protocol given above, reflects the same problem.

Clearly such a misunderstanding was not predicted by the author of the instructional text, and there is no easy way for the learner to check their understanding and debug their model.

One process which exacerbates this problem, is that learners are looking for ways of explaining discrepancies between what they expect, and what happens. S14, whose behaviour was described above, immediately assumed the text was wrong, and changed it, further exacerbating the problem. She never questioned her assumption that the whole instruction was entered into one location: indeed, this assumption may not have been explicit enough to be questioned, and so when the discrepancy occurred, she was sure that the text was wrong. Seeking confirming evidence, and assuming it even if it is weak, is quite common, especially when working with computers. There is a lot of anecdotal evidence of users being convinced that the machine is behaving inconsistently: that their behaviour is exactly the same as on a previous occasion when the computer behaved as predicted, when in fact they are overlooking some action they have failed to perform, or have performed incorrectly. The point here is that the model is very strong. Had there been no intervention, the subjects would have written this off as a problematic experiment and continued. When they next encountered two word instructions they may have realised their error, or thought that the text was wrong here too!

Example 3, Experiment 17, the JUMP and JUMP IF ZERO instructions

Exactly the same problem happens in experiment 17. Two protocol extracts are given, one from S14, and one from S15.

	Experiment book text		**Verbal Protocol**
1	Prepare your microcomputer for use, and put it into the instruction mode if necessary.		
2	Select the location with address 50.	1	Select the location with address 50
3	Put the two word instruction JUMP 20 into your microcomputer by pressing the key labelled JU, then the increment key, then the keys 2 0, then the enter	2	and now put the two word instruction jump 20 into the computer by pressing the key labelled JU, then the INC key, then 20, then enter
4	Check that the instruction is in locations with addresses 50 and 51, its code is 128	3	I don't know if it was automatically meant to put it in both locations, I'll have to check back and see what that's meant to mean.
		4	Right, I've put that instruction into location 51 as well.
5	What do you expect the contents of the program counter to be after the microcomputer has executed this instruction?	5	Emm, what do I expect the contents of the counter to be after the computer has executed this instruction? Eh, I don't know, I presume, er 70, or if it's carrying out both together, 50, or it would be 70 or 70 or 71 sorry. I'm not sure what it means.
	How can you examine the contents of the program counter?	6	As far as I know, the contents of the program counter are 128 in both locations.
		7	I've just checked up on that and they should be 20, um, but I only found that when I checked the answers but I think I can see

Note 128 is the code for JU

Experiment book text	Verbal Protocol
	why they should be that.
	8 And as for how you can examine the contents of the program counter, they're displayed immediately that you've made the computer carry out the instruction.
6 Ensure 50 is in the program counter and then press the single step key	9 Right section 6. 50 is in the program counter. Damn, I've pressed the single step key and I've got error..er (presses some keys). I don't know why this is doing that, (reads) "is in the program counter".
	10 I've gone back to 50 (step 6), I've checked that twice, and it's still showing error. I'm going to go back over it.I've just gone back over it.
	11 Well I think the instruction is in locations with addresses 50 and 51, its code is 128.
	12 First time round, what I did was I took location 50 and put the instruction jump 50 into it by doing JU, inc key and then the keys 2,0. But I also did 51. I pressed 51, entered it, put the JU 20 increment etc in it. I found that confusing and of course it wouldn't carry out the instructions when I wanted it to.
	13 So anyway, I've gone back and when I checked in location 50 the contents were 128, the code for jump, and the contents in 51 were 20, jump 20. In step 6 I checked that and pressed the single step key and the program counter value was 20 as it should have been so that's cleared up but I don't think it's very clear at all.

Protocol extract 7.5: S14 working on experiment 17

The first problem is step 4 where the text says: "Check that the instruction is in locations with addresses 50 and 51, its code is 128" and she says she's not sure whether the instruction has been put into both locations. In fact it has, as in step 3 she made the right move by pressing JU, then INC, then 20, which puts the two words of JUMP 20 into adjacent locations. At step 4, uncertain of whether she's done this correctly, she says she will have to check back, and having done this, says she's now put the instruction into location 51 as well (see protocol segment 4). This has the same result as was seen in the previous protocol: the first word of the instruction ends up in both locations. Step 5 asks her to check the contents of the program counter, and there is a confusion between the program counter and the contents of addresses 50 and 51. When she says the contents of the program counter are 128, 128 may be in recent memory from step 4.

Step 4 says: "check that the instruction is in the locations with address 50 and 51: its code is 128". This could mean that the instruction in both locations should be 128. Segment 6 of S14's protocol suggests that this is how she interpreted it, although she is also confused between the value of the program counter and the instructions she has entered (see segment 5). However, later (8,9), she refers correctly to the program counter. On trying to run the program, (9,10), she gets an error. What has happened is that she put the code for JUMP in both locations which is wrong but she believes it to be correct when she reads line 4, and misinterprets it. (Initially she correctly put JU 20 into locations 50 and 51 ending up with 128, the code for JU in location 50, and 20 in 51, but on reading step 4 she believed this to be wrong, as she thought 128 should be in both locations, and changed it). She finally realises she is wrong and corrects it.

Another subject, S15, whose protocol is given below, also misinterpreted step 4. The text hasn't been included this time, as it has already been given in protocol extract 7.5 above, but where the protocol refers to specific steps these are given in brackets. The protocol segments themselves are numbered, as before.

Verbal Protocol

1 (2) So, select the location with address 50. (3) put jump 20 in. (Explains to the experimenter) - I'm in the instruction mode. You have to press enter before you press any of these keys, otherwise it doesn't work.

2 So you increment, 20, then the enter key (3), check the instruction is in locations..(4) its code is 128, so we can check 51 just be pressing enter, yes?

3 (E) Yes

4 But it's not, it's 30 now.

5 (E) But it's got two bits to it, it's got the jump and the 20 and it's divided into two...

6 But no the code should be 128 and it's 20

7 (E) for the jump, so if you go back and look in address 50

8 (Presses key) Yes, but

9 (E) So the jump is coded as 128

10 But it doesn't say that though, it says here, the way I read it it should be 128 in both, - that's what I expected and therefore I assumed I was wrong and I would keep on trying to do it.

Protocol extract 7.6: S15 working on experiment 17

S15 has called the experimenter in who explains that JUMP is a two word instruction. S15 perseveres in her belief that the code 128 should be in both locations (2.4 and 6) and it is clear that this is how she understood step 4 (10)

7.6 DISCUSSION OF STUDY 2

The problems encountered by PT501 students fall into all three groups: programming, instructional and affective. Whilst many of the problems are in the programming group in that the subjects did not understand how to use an instruction or what it meant, there are more examples of instructional problems where the instructional material acts as a barrier between

the subject and the langauge she is trying to learn rather than a facilitator. The affective problems, are, of course, in the subjects' lack of motivation, and consequently their attrition.

All the examples that have been given are of a conflict between two different states or events. The first is the subject's expectation of what will happen, based on her understanding of the information she has read so far, and the second is either what in fact does happen, i.e. what happens on the display, or what is in the program counter, or what the subject subsequently reads in the text, and therefore believes has happened.

In the first example, experiment 3, there is an additional problem, that of conceptual complexity. The text is introducing indirect addressing, which is difficult, and the core sentence is complex and embedded. Subjects often commented on the difficulty of this text. In a later experiment, S14 commented on a paragraph:

"This sounds more complicated than it really is".

She is referring to the section on how the microcomputer uses the address pointer as program counter. Part of the problem, therefore, seems to be the lack of simplicity: although there is an attempt to make a clear address/contents distinction and to signal it, this is not entirely successful. Although du Boulay, O'Shea and Monk (op. cit.) quote this microcomputer as an example which uses the principle of simplicity, the analysis in chapter 4 suggested it was lacking in this respect, and here is further evidence of this. The kind of ambiguity which was seen in protocol extract 7.5 runs counter to the notion of simplicity. It is very hard, if not impossible, for instructional texts to be completely unambiguous as each learner will involve his or her own knowledge in forming an interpretation. Yet it is important in a text like this, which guides learners in a detailed step-by-step way, to be as unambiguous as possible.

In all three examples, the subjects misinterpreted the text: the text "cues" an inappropriate interpretation. Once they have gone wrong, subjects spend some time floundering before they recover. There are places (step 6, exp. 3, see protocol extract 7.1) where more feedback could be given which would at least enable subjects to know whether or not they're wrong.

Given these problems it is almost surprising that the subjects worked through the book quite successfully. They completed most of the exercises, and got them right: it is only when their behaviour is studied at a detailed level that the frustrations and problems described above emerge.

7.7 CONCLUSIONS AND IMPLICATIONS

The second study supports the suggestion from the first study that the difficulties in working through the PT501 experiment book occur early on. It also provides further evidence that learners have difficulty in developing accurate mental models of how the PT501 machine works: indeed there is evidence that at times the text cues inappropriate expectations and therefore facilitates the development of inaccurate mental models.

The detailed study does suggest why the subjects in the first study gave up. Once subjects have gone wrong by constructing an interpretation of events which is incorrect, and not borne out by later events, the environment is not one which facilitates recovery. The detailed hand holding style encourages learners - indeed requires them - to have a great deal of confidence in the accuracy of the text. In order to recover from their difficulties they need to have confidence in their own judgment, and be convinced that the text (as they have interpreted it) is wrong. Yet there is no easy way of telling which bit is right and which bit is wrong, once difficulties like this occur. For example, recall that S15 in experiment 16, having wrongly interpreted step 8 (see example 2) expected to find the wrong code in address 25 - and crossed out the 1111 1111 in the text, and replaced it with 0110 0000 and thus compounded the problem. In a sense she was right to feel convinced that there was a mistake, but her diagnosis of where the mistake occurred was wrong, and there is little help for students who get into this situation. It is likely that the subjects who kept going were those who were able to cope with such contradictions, in the hope that the next experiment might be clearer. It can be concluded that it will be hard for many novices to become competent at this level when there is no

facility to help them recover.

However, even if such misinterpretations did not occur, there may still be problems in working at this level. The experiment book adopts a step by step "recipe" approach. Do this and do that, and check your results. It is certainly not trying to do this without developing the learner's understanding, and as we saw in the first example, one of the problems comes in explaining indirect addressing. But the experiments, nevertheless, consist of very structured steps which can be followed and can yield the right result, without a deep understanding of what's going on. They work at quite a detailed low level.

Yet as we saw in chapter 2, there is evidence that an important aspect of learning programming is acquiring plans. The PT501 instructions are never presented in terms of plans. The three examples which have been given deal with selecting a location in the program memory and putting an instruction into it (example 1, experiment 3); the LOAD and EXCLUSIVE-OR instructions (example 2, experiment 16) and the JUMP and JUMP IF instructions Example 3, experiment 17). The first example is from an early point in the experiment book before the programming instructions have been introduced. The second two examples, however, are typical of the experiments in this section of the book, which is section 5, the instruction mode. There are seven examples in this section, and all are concerned with introducing one or more instructions. How these instructions might be used to achieve some goal, is not described in detail at this point, though it is mentioned. For example at the beginning of experiment 17 (JUMP and JUMP IF ZERO) the text states:

"When the microcomputer executes an instruction, it increases the program counter (by 1 or 2 as appropriate). This means that if a list of instructions is entered in consecutive locations then the microcomputer can execute them automatically one after another. Sometimes, however, it is useful to be able to make the microcomputer deviate from this list. The JUMP instruction is designed to do this.....
The JUMP IF ZERO instruction only causes a jump to occur if the contents of the accumulator are zero. ....."

In chapter 4 it was seen that the emphasis of the coneptual model is on both state and procedural descriptions, and this lack of a higher level functional view is consistent with the problems that novices had. An alternative approach would be a goal or problem oriented

approach where instructions are introduced in the context of solving a particular problem, and thus their functions would be emphasised in considering questions such as: "In what circumstances is JUMP IF ZERO helpful?" and: "What group of instructions will do the job required in problem X?" The current approach is not conducive to building plans: the steps of the experiments are very detailed and they don't refer to the purpose or goal of the particular experiments, so that once an experiment is started it's easy to get lost in the detail and forget where one is heading.

Another problem with the conceptual model is that athough it was designed to be accessible to novices by virtue of being simple, it is not simple enough. The address/contents distinction which is important in learning a low level language is blurred, and not signalled clearly enough. There is also a lack of feedback to the subject about what state the machine is in. Having made an entry, the subjects had insufficient guidance on interpreting the display - and couldn't be sure whether they had carried out the correct keypresses.

Although the two subjects were able to complete the experiment book, they certainly experienced problems arising from their misinterpretations of the text, and like the subjects described in study 1, they had affective problems in that they did not find PT501 motivating.

Finally, it is interesting to consider whether there are any similarities between the difficulties encountered by the subjects learning PT501 which have been discussed here, and those encountered by the subjects learning SOLO discussed in chapter 6. Although the subjects were learning very different languages which also had very different curricula, there are similarities in some of the problems that they experienced. The behaviour of both SOLO and PT501 programmers is consistent with Lewis and Mack's account of abductive reasoning where subjects generate hypotheses to account for one or more observations. In both studies the subjects attempted to make their own sense of the material they were learning by setting up hypotheses to explain the events that happened, and there are examples in both studies where the text "cued" an inappropriate interpretation. One implication of this is that designers

should be aware of a possible paradox where novices are concerned. They are both active learners who interpret, speculate and hypothesise, and at the same time they place great reliance on the text! In a distance learning situation this is their main, (and perhaps only) source of information and so they scrutinise the text carefully if subsequent events do not bear out their hypotheses to try to resolve the contradiction. Unfortunately, they often seek confirming rather than disconfirming evidence, which is consistent with the behaviour of learners in other problem solving domains.

It is also valuable to consider the differences in the problems encountered in these two domains. The PT501 subjects did experience more difficulties with the domain than the SOLO subjects and sometimes worked through experiments without understanding the process. There is no evidence of this "surface-level" processing in the SOLO study. It is likely that this difference reflects both the different nature of the languages, with PT501 being a low level language, and the different curricula, with PT501 adopting a more recipe like procedural approach. In PT501 the text leads students through every key press and this can be related to the kinds of difficulty that subjects encountered when there was a conflict between their understanding of the text and the model that they had constructed (as in protocol extract 7.6). SOLO's conceptual model was considered to be better than the PT501 model, and this was borne out. There is no evidence of flow of control problems because the subjects did not get to this stage! Their difficulties arose earlier in the course and were connected with understanding and using the PT501 instructions.

Another difference is that many of the PT501 subjects found the language and the experiment book less motivating than did the SOLO subjects, and this increased the attrition rate. To overcome this problem, it was decided to study learners using a different microcomputer system and environment which had benefitted from our experiences of students learning to use the PT501 microcomputer and its follower, the microcomputer for the course PT502. This study, of learning to use DESMOND, is reported in chapter 9.

Chapter 8

LEARNING LOGO

Contents **Page**

8.1 INTRODUCTION

Chapters 6 and 7 discussed the problems which learners have in learning a high level language (SOLO) and a low level language (PT501). In both languages students encountered problems which were to do with the *programming domain* (such as the use of control statements in SOLO and understanding the PT5021 instructions) and also *instructional* problems arising from their interpretation of the text. In both languages there were examples of the text "misleading" the learner and of the learner often developing mental models which were inaccurate. In PT501 there were also problems related to the low level detail at which students were working: following very structured steps. The studies reported here and in chapter 9 have the same aim as the earlier studies reported in chapters 6 and 7: of investigating learning a high level language followed by a low level language and vice versa, concentrating on the transfer of low level skills. LOGO was the high level language studied, and the low level language was incorporated in a small hand held microcomputer called DESMOND (Digital Electronic System Made Of Nifty Devices!!). Students learnt a little machine code, but mainly assembler. There was no strong evidence of any transfer, and this result is discussed here and in chapter 9 which will also deal with subjects' experiences with DESMOND.

Chapter four argued that Logo can be viewed in two ways: as a programming language or as a tool for thinking or problem solving. These two views are reflected in teaching and learning Logo. The first view emphasises programming features, many of which are shared with other languages; concepts such as variables, iteration, recursion etc. The second view emphasises the philosophy behind Logo. This was discussed in chapter four and can be summarised as an open, exploratory approach to learning,

whereby the learner tries out ideas, experiments and modifies these ideas. Bugs are an inevitable part of the process, which help to make the errors in the learners' models manifest.

These two aspects of teaching and learning Logo have implications when it comes to discussing learners' errors and problems. For example, the Logo-as-another-programming-language approach might view a learner's problem with variables as a defect in their programming knowledge (and would ideally therefore intervene to correct it), but the Logo-as-philosophy-of-learning approach would view such problems as an inevitable (and welcome) part of the learning process, where no intervention is desired. For this reason, the chapter is treated rather differently from chapters 6,7 and 9. A section on the curriculum is given, because much more than the other languages, behind the Open Logo (OL) tutorial manual is a philosophy of learning, i.e. the second approach. The implications of this need to be discussed in order to make sense of the subjects' experiences and probelms. Section 8.1, therefore, introduces the particular curriculum which subjects followed to learn Open Logo (BBC, 1983) and gives an evaluation of the Open Logo (OL) tutorial manual. The remainder of this chapter describes the study and presents the results.

The curriculum: the Open Logo tutorial manual

This section describes the Logo curriculum used in this research, i.e. the Open Logo tutorial manual. (There is also a reference manual for Logo, but this was not used in the study).

Aims

No overall aim is given but the tutorial manual:

"is intended for people with little or no experience of the language Logo. It is divided into sections which either introduce new Logo commands or illustrate basic principles of Logo programming.

Everything new is introduced in the form of an example, and you the user are invited to try out each idea as you build up your own repertoire of Logo programs."

Order of teaching

The tutorial manual is divided up into 7 sections. There is a gradual introduction with exploration of the various commands and drawing geometric shapes in section 1. Section 2 introduces curved lines with the commands ARCL and ARCR and freehand drawing. Section 3 introduces procedures - and the commands used in sections 1 and 2 are incorporated into procedures. Section 4 introduces several important ideas: procedures with variables, modular programming and passing values of variables from procedure to procedure. Section 5 deals with recursion, starting with tail recursion and progressing to total recursion. It also includes conditional statements, the MAKE command, list processing and dynamic lists. Section 6 introduces the Logo calculator and the arithmetic of variables and the commands ASK, RUN, SAY and DISPLAY. Finally section 7 deals with user defined functions and conditional expressions in functions.

Pace and content

The pace of the first 3 sections is quite slow in that they occupy 44 pages and take students as far as procedures: the development thereafter is much faster. The beginning of section 4 introduces variables, starting with single variables and moving to the use of several variables in 4.2. Section 4.3 introduces modular programming, procedures within procedures, and 4.4 introduces passing values of variables from procedure to procedure, using the petal, flower, and border example where the procedure FLOWER uses PETAL and BORDER uses FLOWER. The passing of the values from one to the other , however, is not discussed in any detail. Shortly after this tail recursion is introduced and then the previous two sections are combined in an

example where the value of a variable is changed on each recursive call. Lists are then introduced - as the values of variables using a procedure called TO WANDER. WANDER is then amended to deal with the empty list problem and the next procedure, CYCLE uses WANDER. Dynamic lists follow, and shortly afterwards total recursion is introduced, (see Q5.3 and Q5.4 in Appendix 8.2 for the example). One further example is given. The final 12 pages are taken up with the Logo calculator, arithmetic of variables, user defined functions and conditional expressions in functions. The concepts which are introduced, and when they are introduced is given in appendix 8.1.

Style of teaching

The teaching style takes two main forms: direct instruction and invitations to explore and make predictions. The learner is told how to start up Logo and what to expect through direct instruction:

"Respond by completing the line
*Logo and then press RETURN. Now you will see the Logo message on your screen and Logo will be running." (p4)

The learner is told to type in specific instructions:

"type the Logo instruction
PRINT 'HI.....Logo will respond by typing HI ....." (p6)

After a short section on PRINT and correcting mistakes, COPY etc, the manual says:

"Now it's your turn. Get Logo to print out your messages using the PRINT ".....instructions." (p8)

The same style and tone is continued when the turtle is introduced (section 1.3) but soon predictions are invited:

"Type in the following sequence of instructions....Can you guess what the turtle is going to do in response to each instruction?"

FORWARD 100
RIGHT 90 ...."(etc)

Screen drawings are given of the effects of these commands, and again, exploration is invited:

"You should now be ready to use the turtle to thoroughly explore turtle space and at the same time produce exotic drawings" (p16)

The aim seems to be to introduce an idea and then for the learner to explore that idea. For example, the instructions for a square are introduced - and the idea of using the COPY key to repeat the forward left 90 procedure (although discovering these two steps was invited by a question) and then the learner is invited to try the same idea with different lengths and angles:

"You ought to try repeating the exercises using the same instructions of your own, for example repeatedly using.....FD 200 LT 144 gives a pentagram."

A little further on, the author takes on the role of fellow learner, as in the following answer to an in-text question:

"I found that I could draw .. a pentagram using REPEAT 5 [FD 250 LT 144]"

This is a common device in Open University texts. It avoids the problems of saying: "The correct answer is....." The co-learner role is continued, and this style and role is now adopted throughout, mixed with direct instructions, e.g:

"Use the commands.....to change the pen.....I prepared the screen.....and then drew the picture below."

This style suggests that the author has been influenced by the exploratory discovery learning approach which is often associated with Logo, although there is little suggestion of the learner making "new" discoveries: rather the author is reporting the results of hiw or her own pseudo-"discoveries". Furthermore, the philosophy behind this is not made very explicit. For example, there is only a brief mention of the idea of bugs as positive feedback, even though the idea of welcoming errors is likely to be new and alien to adult learners.

The examples used

Most of the examples in the tutorial manual either involve turtle graphics or mathematics. Turtle graphics is the obvious starting point, and various shapes and drawings are constructed: including spirals and borders of flowers. List processing uses mathematical examples, and the recursion examples again are mathematical or geometrical, the only example of total recursion is a pattern within a square, at each corner of which there is a similar pattern, only 1/3 the size. Sections 6 and 7 contain only mathematical examples. In section 6.1 this is understandable, as it is the section on the Logo calculator. Section 6.2 is the arithmetic of variables. An example here is:

Instruction	Response
e.g. PRINT JOIN C C	[1 2 3 A 1 3 2 A] (p77)

Section 7 looks at the RESULT command and the main example is drawing a right angled triangle:

```
RTRIANG BAS HGT
RT 90
FD BAS
LT 90
FD HGT
```

```
LT EXT BAS HGT
FD HYP BAS HGT
```

where the functions EXT and HYP are defined:

```
EXT A B
RESULT 90 + ATN(A/B)
```

and

```
HYP C D
RESULT SQRT(C*C + D*D)
```

The role of in-text questions

In the Logo tutorial most of the questions have a teaching role: they lead on to the next point by inviting the learner to make a prediction, or discover a problem, write a procedure or explore what has been introduced. The questions are given in appendix 8.2. It is often through these questions that the open, exploratory style is carried. However, there is a problem in how this sits with the rest of the text - and with learners' expectations. For example, question 1.1 has a flavour of "Guess what I'm thinking of" and questions 1.3 and 2.1 leave the learner wondering how long they should keep trying to draw a circle or a petal. Of course, the "How long should I try?" question arises with DESMOND too, but the exercises are less open in that usually the learner is being asked to create a program to do a specific job.

As with DESMOND, the questions given to the students were the in-text questions as the intention was to use both curricula as they stood. Unlike DESMOND however, the nature and relative sparseness of the Logo questions means that the answers to them are not such good indicators of the students' progress.

8.2 EVALUATION OF THE TUTORIAL MANUAL

This section summarises the views of three educational technology experts about the manual's likely effectiveness. Their opinions were sought following the study, as subjects were sometimes quite critical of the curriculum, as expressed in the tutorial manual. Unlike the other curricula, the Open Logo tutorial manual was not developed specifically for a course, and so had not undergone the detailed field testing which the other curricula had. On inspection and working through the manual before recruiting subjects no major problems were apparent, although there were some inaccuracies which meant that errata sheets had to be included. A copy of the tutorial manual was sent to an experienced Logo teacher who also approved it. However, during the study it became clear that many of the subjects' problems were related to the curriculum. To gain a more objective view than the researcher's (whose judgment was probably affected by the subject's problems) three educational technologists were asked to comment on the tutorial manual. All of them were experienced programmers and had taught novices programming. Two had extensive experience of research in Human-Computer-Interaction, and the third was a researcher in the area of novice programmers. An overview of their comments is given next.

Evaluators' comments

Of the three evaluators, only one predicted the problems which the subjects in fact reported, and this was the person who was a regular user of Open Logo (OL), and had previously looked at the manual with a view to using it. He was the most critical of the three. Given that the tutorial manual had been assessed by two other people previously (the researcher and the Logo teacher who was consulted before the start of the study),

this means that only one out of five people foresaw problems. There are two possible reasons for this:

1 that the shortcomings of the tutorial manual are not too serious (i.e. this person was being especially critical);

2 it is very difficult to predict the kinds of problems novices will have with instructional material, and the only person who did point to the kinds of problems that novices did have, had extensive experience of using OL.

The data from the study, discussed in the following sections of this chapter, suggests that some of the features of the OL manual, and the inaccuracies, although they may not be serious flaws, did cause problems for novice learners who were working on their own. The experience of several of the staff at the Open University Institute of Educational Technology, where this research is based, suggests that the second reason is also true: it is extremely hard to second guess the problems that novices will have. The three evaluators' overall views are summarised briefly, before looking at the more detailed points that they made.

Evaluator 1

This person's reaction was positive. She liked the "friendly reassuring tone" of the "tutor", but thought the text lacked global explanations at the beginning of some sections, and also a quick introduction to the "Logo machine". She wondered whether lists were introduced a little late, and commented that recursion "is just not easy to explain".

Evaluator 2

This person liked the introduction to recursion, but was was more critical about parts of the manual that she felt may cause difficulties. The general points were:

1 The differences need to be explained between commands that appear to be similar, e.g., ERASE, DELETE, CLEAR, BLANK, etc. It would be helpful to include a summary, giving a "model" for each command, and grouping them according to function.

2 Some of the terms in chapter 7 need to be explained to students with a non-mathematical background.

Evaluator 3

This was the most critical report. He makes two main criticisms.

1 The writer does not communicate well and communicates with the wrong person:

"Firstly, it is not clear whom the intended audience was; whether it was school children learning Logo, or adults (Open University students perhaps). In some places I found it patronising, which suggests that it was written with children in mind, but even so would be unacceptable. The author should be told that putting an exclamation mark at the end of sentences does not make them chatty and friendly. ...it merely annoys.

...the writer has often made assumptions about the meaning of words. In his..world some words have quite specific meanings which are well understood, but they do not have the same meaning in common speech."

Part of this style of communication that is disliked is the anthropomorphism.

2. The second main criticism is that the text contains inaccuracies and the implications of this for novice learners[1]:

[1]Some of these inaccuracies are not errors as such, but places where the text is ambiguous. These were not included in errata sheets.

> "novices tend to expect the same fidelity they would find in the printed word elsewhere. An inaccuracy in a text for novice users can therefore be most damaging. The reader is likely to lack confidence in their own ability and judgment and if anything happens on the computer which does not correspond with that which the text told them to expect, then they will assume it is they who are at fault; they would not believe the text could be wrong"

The remainder of this evaluation is in two parts. First some of the more detailed comments made by the three evaluators on chapter one of the OL tutorial manual are given, to illustrate the kinds of comments they made. There are less comments on the other chapters, and one evaluator thought that the first chapter was the worst. Following this, examples are given of points made by the evaluators which correspond to problems that students reported. Complete evaluation reports are given in appendix 8.3 .

Comments on chapter one

Text	Comment
...if there are chips with a higher precedence"(Page 9)	This phrase will either be meaningless to a novice, or worry them.
>*Logo	It may not be absolutely clear that the user does not type the >. Also, the text assumes no programming background, but talks about moving back and forth between Logo and Basic.
PRIMT 'HO	The previous example was PRINT 'HI. The text does not specifically point out that there are two deliberate errors, so if only the HO is noticed (and not the PRIMT), the instructions that follow become confusing
"that is a list to be quoted" (page 11)	This would be meaningless to a novice

Inconsistent use of terms "diagnostic message" and "error message"

Evaluators' comments and students' reported problems

The educational experts were asked for their opinions in order to try to answer the question "Were the learners right?" Was the tutorial manual a "faulty" piece of instruction, or did the learners expect learning programming to be too easy? The evaluators' reports suggest that at least some of the learners' comments were justified in that some of the problems they predicted were indeed encountered. These include how brackets appear on the screen, the BBC microcomputer's eight different modes, the meaning of commands such as COPY and WIPE, the significance of ends of lines etc. A full account of these is given in appendix 8.3, but a few examples are given below.

Text	Comment
The BBC microcomputer has eight distinct modes.	The term "mode" is not explained, and is used in different ways. There has already been mention of teletext mode, and there is later mention of text mode and drawing mode; however, these are not part of the eight distinct modes, so this is very confusing. As this is introducing MODE 1 they could be simply told to type it here.
BD 50 LT 90 FD 50 RT 90 (page 25)	One of the problematic features of Logo pointed out by one of the evaluators, is the significance of the ends of lines. The authors of the OL tutorial manual have dealt with this by trying to show explicitly where ends of lines occur. This may cause some confusion, however, because readers may think that

where the end-of-line symbol appears they must type a carriage return and this is not always true. This statement is an example of the problem of ends of lines. None is shown on this line and none is necessary, but the reader may wonder how the next statement got on the next line without a carriage return. In fact, they would have to type a large number of spaces to achieve that. The same is true in the example in the following page, where no carriage returns are shown, so strictly speaking, nothing would happen.

"Now you can use the TO command and the editor to construct programs for any of the procedures you have already met. For instance type TO STAR R and then use the editor to construct the program

```
STAR    R                -program STAR
PENDOWN R                -draws
REPEAT 5[FD 50 RT 144] -  5 lines"
```

When the editor is opened to create a new procedure the name of that procedure appears on the top line. The user does not need to type it. The example given here is the first instance of this happening. The text implies that the user must type the name STAR, and this is reinforced by the fact that the line containing the word STAR is shown followed by a carriage return character (R). (The carriage return must be there but it is inserted automatically by the editor, not typed by the user.) If the user types everything shown in the example, there will be two lines containing the name STAR, and the user will have accidentally defined an infinitely recursive procedure. In fact, when subjects tried out STAR, their program usually ended up looking like this:

```
TO STAR
STAR
PENDOWN
REPEAT 5 [FD 50 RT 144]
```

This confusion is also present further on (e.g. p 42).

8.3 THE STUDY

Subjects were given a Logo tutorial manual, and the Logo chips were installed. Some people had their own microcomputers and therefore had constant access. For others, access was arranged to BBC micros on the university campus at agreed times, usually lunchtimes and evenings. Arrangements were also made for the micros to be borrowed over weekends.

Design

Twenty people began the study, and were allocated to two groups. Group 1 worked on DESMOND first followed by LOGO whilst group 2 worked on LOGO and then DESMOND. Most of the subjects were female: with one man in group 1, and two men in group 2. The number of subjects was limited by hardware availability.

Subjects

All the subjects who took part in this study were Open University employees. They were recruited via an advertisement in the University's in-house magazine and were told that to be eligible they should have no programming experience. Information was collected about their academic qualifications, but they were not selected on this basis.

The task

Subjects were given the OL tutorial manual and were asked to work through it, and to attempt all the in-text exercises (see appendix 8.2) and to fill in comments sheets. They were asked to explain any difficulties they were having and to give answers to the in-text exercises and problems. Interviews were set up every two

weeks for pairs of students and they were told that the work they had done the previous two weeks would be discussed at these 'interview' sessions. The interview sessions were not always held at the planned intervals: those that did not were mainly because people had got stuck, and therefore had not completed what they intended to. In these cases either the interview was delayed, or more commonly, the experimenter and subjects met to discuss the problems they were having and sort them out and an additional time was fixed for another meeting. At the interviews the subjects were asked about any problems they had, and the in-text questions, and were set additional questions. Because of the nature of the in-text questions, i.e. that they had a teaching function, they were not taken out of the tutorial manual. Subjects were encouraged not to 'cheat' and there is no evidence is that they did.

8.4 RESULTS

Two kinds of results are discussed here. The first is the syntactical problems reported by the subjects and their difficulties in understanding error messages. The second set of results is the subjects' performance on the in-text exercises and an analysis of the errors. These are programming problems. Particular problems that the subjects encountered will then be discussed in more detail in section 8.5.

Reported problems and error messages

Syntactical and low level problems

The syntactical problems and also the error messages reported are shown in tables 8.1 and 8.2.

Didn't know had to enclose statements greater than 1 word in brackets (1)
Forgetting that (RETURN) is not part of the command (4)
Forgetting about CLEAN - using BLANK instead and losing commands (1)
Needed to know about BLANK earlier (1)
Can't find out how to use PAINT (3)
Short reference list of commands would be helpful (3)
In re-editing WANDER don't know where the line breaks should be (2)
What is "turtle?" (4)
Doesn't always print brackets for brackets (prints < > instead of [ ] when not part of procedure) (7)
What is a primitive command? (2)
Every time I had brackets plus or multiplied by something else it didn't give the expected answer (1)
Used square brackets instead of round as so used to them, - it wasn't pointed out (2)

Table 8.1: Reported syntactic and technical errors (Number of times reported in brackets)

Error messages	Category	Reason for error
Out of space in procedure STAR (10)	1	Accidentally wrote
Out of space in procedure SQUARE (1)	1	recursive procedure
Out of space in procedure HEX (1)	1	
Out of space in procedure TEST (1)	1	
Out of space in procedure BOX (3)	1	
Out of space in procedure POLYGON (1)	1	
Out of space in procedure BORDER (1)	1	
Out of space in procedure RECUR (6)	1	Infinitely recurring procedure
REST has no meaning in procedure WANDER (4]	2	No value for REST, BOX, SIDE etc
BOX has no meaning in procedure...(2)	2	
SIDE has no meaning in procedure.. (1)	2	
SQUARE has no meaning in procedure..(1)	2	
BUT has no meaning .....(1)	2	
SPIRAL has no meaning in procedure.. (2)	2	
A has no meaning in .. (1)	2	
BOB has no meaning... (1)	2	
AC has no meaning ...(1)	2	
Please switch the machine off (2!) and on again	3	Not sure!
Don't know what to do with 200 in procedure BOXES (2)	4	Unsuccessful use of functions
/ expects numbers on its left and right in procedure RECUR (1)	5	As stated in message
Empty LIST or STRING FIRST failed in procedure WANDER (3)	6	As stated in message
Numbers too big in procedure RECUR	7	As stated in message
Number expected Repeat failed	8	As stated in message
TRUE or FALSE expected make failed in procedure SUMSQUARE (1)	9	"

Table 8.2 Error messages and diagnoses

There are two issues of concern about both of these. The first is that the Logo manual had a small number of typing errors - and needed errata sheets.[2]
The second point is that there were parts of the manual which weren't particularly clear: for example in emphasising the use of RETURN:

"When I did R on all these my computer complained so I thought that perhaps R was somebody making the point that perhaps some people were forgetting to put return or stood for return.
But earlier on it had said somewhere be careful to do things exactly as they are written including spaces, so I did."

This problem is related to logical and physical lines, which was mentioned in the evaluation section (8.2) but is never explained in the manual. Unfortunately, such problems have a larger effect on novices in a distance learning situation than is likely for experienced users with advice close to hand. When novices are asked to follow the text exactly, that's what they do - and any mistakes or ambiguities tend to be magnified. In situations where less close attention is demanded, we might expect someone to reason about whether a piece of text is inadvertently faulty. In this case, it makes sense to resort to this explanation only as a last resort.

The discussion that follows concerns the error messages. A distinction is needed between the error messages themselves and why they happen; i.e. their cause. The reasons for getting the messages are considered first. The first 8 messages, categorised as (1) in table 8.2, are all variations on a theme. Out of space in procedure X occurred because of an ambiguity in the text that had far reaching consequences. This is the issue about typing the title line of the procedure which was discussed earlier, and resulted in subjects accidentally producing infinitely recursive procedures, which quickly ran out of space.

---

[2]The version of the tutorial manual used was believed to be the final "handover" version, but in fact it did contain some errors.

There is little chance of a novice diagnosing the reason for this error as they have hardly started working on procedures. The next most frequent error message related to the use of variables. Subjects were working with Logo on chips in the machine and so memory space was limited, and because the programs they wrote tended to be very short and simple, they rarely saved procedures from a previous session. However, if they broke off in the middle of a section - the text assumed all the work done up to that point was in the machine. This sometimes led to this error. Other times variables had not been defined, or the number of variables in the title line did not agree with that in the body of the procedure.

The other messages are more helpful and the symptoms (i.e. the error messages) are more closely related to the cause. One of the most worrying was the third: "Please turn the machine off and on again!". No particular reason was found for this which is not a Logo error but a system error - except that perhaps the BBC micros which were rather old were overheating!

The main problem with the error messages themselves is that except for nos 4 - 8, the most common messages, 1 and 2, are completely unrelated to the underlying cause, the actual problem. Students are simply not in a position where they can diagnose their problem from the error messages. "Out of space in procedure N" serves only as a general indication of the problem, not an explanation of it. For error messages to be helpful they need to explain the problem at a level students can understand - that is in terms of the students' current knowledge. The errors in category 2 failed from this standpoint: for them to be understood, a link would be needed explaining that the variable SIDE or whatever acquires a meaning in procedure SQUARE when it has a value - and that the likely cause of the message is

that for some reason the variable doesn't have a value. These problems confirm the view expressed in chapter 4 that Open Logo's conceptual model is not particularly simple because the error messages are not consistent with the tutorial manual, and are not couched at a level that the novice can understand.

Answers to in-text questions

Some people made no attempt at particular questions. As this often indicates the difficulty they perceived in answering the question, they should not be left out of the analysis. Therefore rather than taking correct answers as a proportion of the total attempts, the number of correct answers is taken as a proportion of the number of people actively working through the course at this point. Where people have dropped out completely, they are omitted from this base figure. Table 8.3 below gives the average number of questions answered correctly by each subject. (There were 19 questions altogether).

Group 1		**Group 2**	
S1	14	S7	12
S2	7	S8	16
S3	7	S9	11
S4	11	S10	14
S5	8	S11	5
S6	14	S12	13
		S13	8
		S14	11
		S15	10
		S16	13
Gp 1		Gp 2	
Average	10.2	Average	11.3

Table 8.3: Average number of questions answered correctly

The next table, table 8.4, gives the percentage of correct responses for each question.

	Group 1	Group 2	Total
Question			
1.1a	33.3	70	56.3
1.1b	16.7	80	56.3
1.2	100	90	93.8
1.2b	100	80	87.5
1.3	100	60	75
1.4	16.7	30	25
2.1	100	60	75
3.1	87.3	70	75
3.2	100	50	69
4.1	83.3	80	81.3
4.2	83.3	80	81.3
4.3	66.7	60	62.5
4.4	33.3	50	43.8
4.5	50	50	50
5.1	16.7	30	25
5.2	16.7	30	25
5.X	0	50	31.3
5.3	0	0	0
5.4	16.7	80	56.3

Table 8.4 Percentage of subjects successfully completing each question

In terms of problems then the most interesting questions are 1.4, 4.4, 4.5, 5.1, 5.2, 5.X and 5.3 (in bold): all of which have a 50% success rate or less.

In terms of attrition the striking thing about Logo, compared to DESMOND (which is discussed in chapter 9), is that no-one dropped out until near the end of the course. The people who did Logo first (group 2) all went on to do DESMOND. Some felt they were struggling at the end, but all completed the majority of the work. However, of group 1, 4 dropped out near the beginning of DESMOND and never went on to study Logo.

Comparison of performance

Table 8.3 also gives the total average number of questions answered correctly by each group: 10.2 by group 1, and 11.3 by group 2. The difference between the groups is not significant, but it is interesting, in that group 2 (learning their first language) performed slightly better, and so any difference could not be attributed

to prior experience. It is possible therefore that group two turned out to be a slightly "better' group.

Errors and problems

Where possible, for programming problems the same categories were used for the Logo and the DESMOND studies, but in fact the DESMOND errors fall into a greater nummber of categories. One of the reasons for this is that there were considerably less programming questions in Logo, and so consequently less opportunity for errors! A complete list of the questions is given in appendix 8.2. There are a total of 48 questions answered incorrectly (not including missing answers) which are distributed as shown in table 8.5.

Question	Errors	Question	Errors	Question	Errors
1.1a	6	3.1	1	4.5	4
1.1b	4	3.2	3	5.1	2
1.3	4	4.2	1	5.X	1
1.4	6	4.3	1	5.2	2
2.1	3	4.4	4	5.3	6
	---		--		--
	23		10		15

Table 8.5: Distribution of errors

The different types of Logo questions are: problem solving (e.g. 1.1 "Can you think of a more systematic way of drawing a square?"); constructing programs (e.g. 3.1), modifying programs (e.g. 3.2) and predictions (e.g. 1.4). Also eighteen of these answers are too incomplete to be analysed, leaving 30 errors to be categorised, and these were divided into four categories. These categories, and the number of successes in each, are given in table 8.6.

1. Missing code/incomplete answer	10
2. Missing or inappropriate plan	8
3 Incorrect prediction	7
4 Incorrect code	5
-----	--
Total	30

Table 8.6: Categories of errors and number of errors in each

Cateogires 1, 2 and 4 are the same as DESMOND's categories 2, 3 and 4, (given in the next chapter) and there is an additional category: incorrect prediction. The other categories used for analysing the DESMOND errors were not appropriate, as none of the Logo errors fell into them. The extra category applies to prediction questions which did not exist in DESMOND.

1. Missing code/Incomplete answer

These are usually answers which fail to fully answer the question rather than answers which contain missing code:, e.g. on Q 1.2(b) : "How many times does each of the lists need to be repeated? Why?" the answer to the second part (why) of "to reach the starting point" was judged as incomplete.

2. Missing plan or inappropriate plan

Plans are two or more instructions used together to achieve a certain result. They are also generic and can be used in various different situations. A typical Logo plan therefore might be a square or a triangle plan. Errors are classified as plan related where the answer is wrong because a plan is missing (e.g. Q3.1 requires a hexagon plan) or inappropriate.

3. Incorrect prediction

These were answers to prediction questions such as 1.4, or 5.3. For example, one answer to 5.3 was: "it will be like SQUARE 200 because of the RT instruction".

4. Incorrect code

This category applied only to the programming problems, e.g. Q 5.2.

The category containing the largest number of errors is incomplete answers or code, category 1. Given that the extreme examples of this have already been taken out because they were too incomplete to analyse, it is clear that the biggest problem for Logo subjects is in producing a complete answer. The second category is plan-related. In Logo, the intention is often for subjects to discover such plans, and the in-text questions are part of this discovery method. (For example, question 1.3. invites the learner to find the commands for drawing a circle). It is likely, therefore, that in terms of the author's aims, these plan failures would not be seen as problematic, but simply as an expected part of the learning process.

This categorisation of errors does not, however, give a full enough picture of the problems subjects had, many of which were instructional and affective. This is because there were fewer programming exercises set in the Logo tutorial manual than in the DESMOND experiment book, and the number of errors given in table 8.6 above refers to errors in all the exercises i.e. including non-programming. Inspection of the exercises in appendix 8.2, and subjects' responses to the non-programming questions suggests that subjects found the questions relatively easy and so their answers do not reveal the kinds of problems the subjects experienced.

The next section, therefore, discusses the problems reported in subject interviews and the comment sheets. The most difficult questions are used as a starting point for looking at the concepts that subjects had difficulty with, and the interview data and comment sheets were also analysed.

8.5 DISCUSSION OF ERRORS AND PROBLEMS

Appendix 8.1 gives a list of Logo concepts, and indicates which questions they relate to and in which part of the curriculum they first appear. Table 8.4 gave the percentage of correct answers to these questions. It can be seen that question 1.4 which is a prediction question testing detailed following of embedded repeats was often not successfully answered. Question 4.4, which draws on using a variable plus the relationship of the sides to the angles also proved to be problematic. Question 4.5 requires the idea of calling up a procedure from within another. Only two of the subjects answered this correctly, using a modular approach. Questions 5.2, 5.3, which concern recursion, were not generally answered well, which supports students' perceptions of their lack of understanding here. Question 5.4 was completed successfully by those who tried, yet they all commented that it was possible to complete it without any real understanding.

To summarise, the explorations of the relationship between the number of sides and the angles of a figure in the earlier section are quite successful. Students do not, however, have a good grasp of how several variables are used and how values can be passed from procedure to procedure, of modular programming, or of recursion, or of lists; and it appears that the last section of the manual passed through undigested. The discussion that follows looks at these areas, at the

mismatch between the potential of the Logo curriculum (as described in the introduction) and the reality of the learners' experiences.

Variables and passing values

An accurate model of variables, how they acquire values and how the values are passed from procedure to procedure is important, and is a pre-requisite for understanding procedures and recursion. The in-text questions which test this understanding are in section 4. Looking back at table 8.4 it can be seen that questions 4.1 and 4.2 caused little problems, but 4.3, 4.4 and 4.5 were more problematic, and so these three questions will be discussed in some detail.

Question 4.3

On question 4.3 there were two unsuccessful attempts, (S23 and S31) which were very sparse, although S31 later solved the problem. However, the comments from those who were successful are also helpful in understanding what was happening here. The question is:

"Can you modify STAR so that it will now have 3 variables: NUM (number of sides), SID (side length) and ANG (angle).

Perhaps you should also change its name to POLYGON to make it more accurate.

POLYGON NUM SID ANG

should be able to draw any polygon (with the right values)"

The problems encountered by those were eventually successful included producing programs which were accidentally recursive (see section 1.2), not knowing where to put brackets, and understanding variables. One subject who was finally successful said:

"At last! (Gives correct answer) I was going wrong by forgetting that a variable was an instruction's value"

Another also had problems, and made the following attempt which suggests that she is misinterpreting the function of a variable as a command:

```
POLYGON NUM SID ANG
    NUM [FD SID RT ANG]
```

Both of these examples indicate a lack of understanding of the basic syntax of using variables and a trial and error approach.

A further instructional rather than programming problem concerned ambiguities in the programming manual, the first example of which is the bracketed text at the end of the question (i.e. "with the right values") which led one person to ask:

"Does it mean that only some have them or that it depends on my ability to get it right?"

The answer, of course, is the latter. Another person (who didn't give an answer) thought there was a problem in that for the procedure to work on any polygon you need to "fix" the variables at the correct value - hence the bracketed comment in the manual. The point of this was to lead on to questions 4.4 and 4.5. However, he comments:

"I thought the value NUM was meaningless as for instance you can't draw a 10 sided polygon with SIDE 50 ANG 60 so I can't see (4.4) how you can draw different polygons without changing the variables SIDE, ANG to the correct values."

Question 4.4:

"Can you further modify POLYGON (or create a new procedure called something else) which has one variable (NUM - number of sides) as its input and will draw a polygon with that number of sides? e.g. POLYGON 4 should draw a square, typing POLYGON 3 should produce a triangle."

The last comment above indicates that for one person at least, question 4.3 misled the subjects, instead of preparing them for question 4.4, which was the intention. On question 4.4 there were 6 unsuccessful attempts, but only 2 of these gave any answer: the rest didn't know how to go about it. Of those who got it right, 3 had not realised that division could be achieved by the '/' symbol, so made comments like:

"Logically this seemed the only way, especially given the hint in 4.2 but didn't expect the computer to be able to do division as part of the command and was trying to think of an alternative."

Four of the subjects made no real attempt. The main difficulty here was how to solve the problem rather than turning the solution into Logo:

"I cannot see how to do this at all. I can make it draw any of the above polygons but only with an ANG variable too, e,.g. FIG NUM, REPEAT NUM [FD 100 RT 90] will draw a square with constant length side. Of course it just repeats one side if I put in FIG 5. FIG NUM ANG, REPEAT AND [FD 100 RT ANG] gives POLYGON but I can't work out how to change NUM."

Another problem, as mentioned earlier, was not knowing if there is a way of doing division or not knowing what it is. The division command had been given in the text, but many subjects had not really taken it in. One subject had worked out this

relationship but failed to apply it because she hadn't realise that the divide facility existed. Her interview extract below shows that she had worked out the necessary algorithm but didn't realise that Logo gave a procedure for dividing:

"When I got on to 4.4 I tried no end of different things.....and I could not work out where..... I mean with the other one I'd had to put 3 instructions in - the number of sides, length of sides and the angles. With this one you were just asked to put the sides in, and obviously if it's going to be a four-sided figure, you're going to have 90 degree angles; ....and so on. If you're only going to have one program in there and to put in polygon 3 ..(or). polygon 10, I couldn't see how I could have that one program that would adapt all the sides and the angles that were needed, and so I tried lots of different things but I couldn't get anywhere near it.....Are you going to give me the answer so I can see where I went wrong?"

[E] "Yes,....you were almost there on the last one, where you'd worked out the relationship between the number of sides and 360 degrees. ...So you can use that"

"Yes, I'd thought of that but we'd had no procedure for divide.....
That was the sort of thing I'd wanted to do...to divide its sides by the angle but I didn't know how I could go about that."

Interview extract 8.1: S26 talking about question 4.4

This problem is similar to one that has been encountered before, in learning SOLO, where examples were given of subjects who were constrained by the models that they had developed. Here, the subject can only see one way to answer this question but is unable to activate this model because she is so constrained by the belief that it is not possible.

One subject did understand the agreement of variables and values, and the relationship between side and angle of a figure. In his first attempt he had declared the variables in the title which were not referred to in the body of the procedure.

Additional questions

The additional questions which are relevant here are 1.2, and 3.1.

Question 1.2 asked the student to draw four intersecting flags:

"Can you construct a procedure (or more than one) to draw the figure below?"

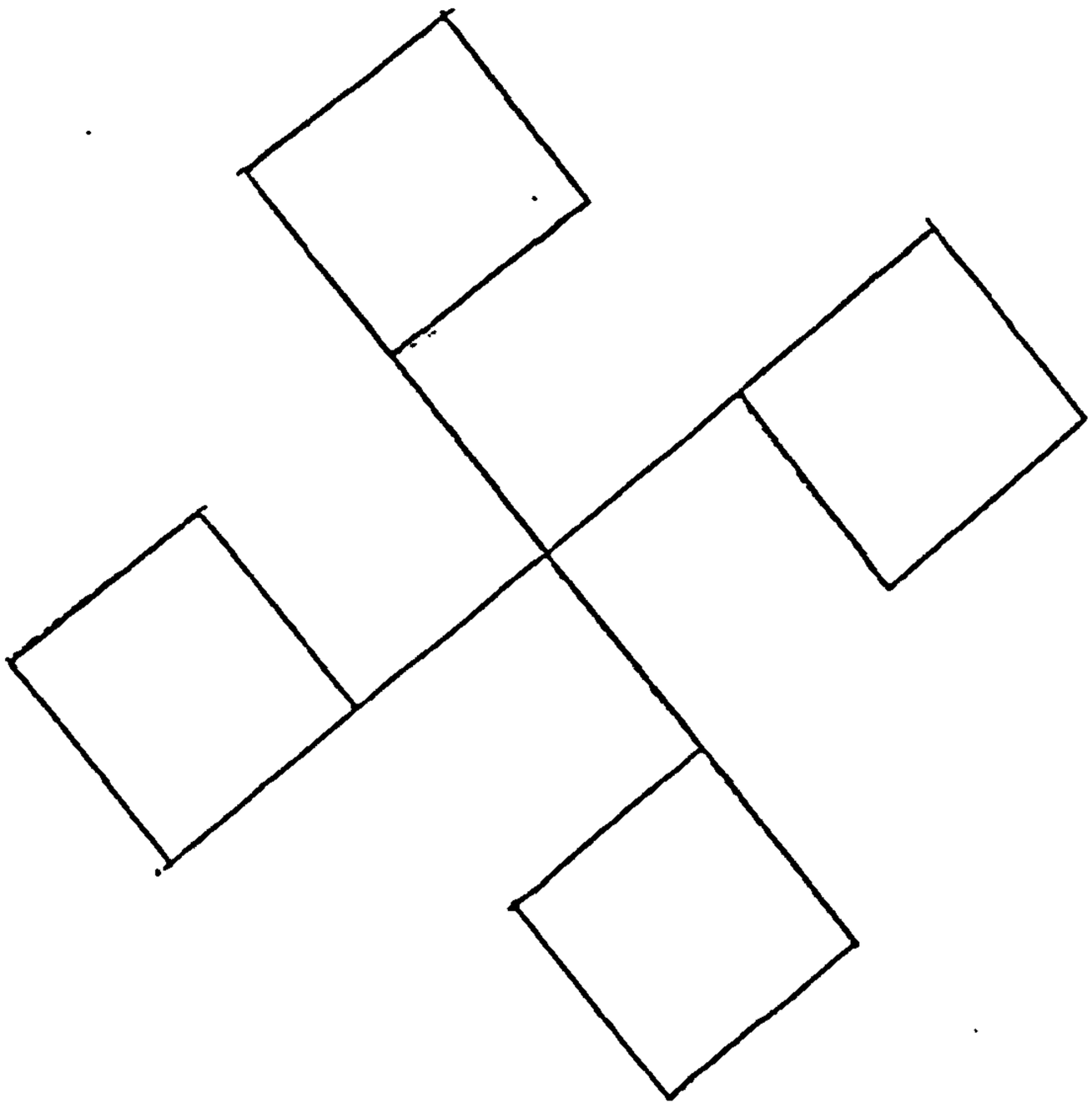


Figure 8.1: Intersecting flags

This question is a replication of one used in Kurland et al's study (Kurland, Clement, Mawby and Pea, 1984). There are only 3 scores here to look at. As the curriculum took longer than most people expected only 3 subjects completed this task which they did at home. All three were successful in their use of variables.

Question 3.1 tests for agreement in the use of variables between the title of the procedure and the body:

"In section 4.4 three procedures were defined:

1. VPETAL SIZ

```
REPEAT 2 [ARCL SIZ 60 LT 120]
```

```
2. VFLOWER SIZ
   REPEAT 6 [VPETAL SIZ LT 60]
```

```
3. VBORDER SIZ DIST NUM
   REPEAT NUM [VFLOWER SIZ FD DIST]
```

3.1 What will happen if you create these three procedures, as above and then type in VBORDER 1000 60?"

On this question there were 2 correct and 3 incorrect scores, which would not suggest a great depth of understanding.

There is considerable variability in students' understanding and use of variables. This section has looked mainly at the difficulties and discussed the following bugs: syntactical, understanding the function of variables, and non agreement between declaration and use of variables. Other problems emerge which are not specific to variables (but nonetheless lead to buggy answers) and include difficulty in finding problem solutions which is constrained by the lack of familiarity with one particular command (/); textual ambiguities and accidental recursion.

Modularity

One of the claims made for Logo is that its modularity encourages decomposition skills, i.e. that problems can be broken down into sub-problems and each tackled separately. However research by Pea and Kurland (1984) found that Logo novices (in this case children) often fail to write modular programs. This section looks at the extent to which the subjects in this study adopted a modular style. One straightforward way of testing this is to see whether procedures are used as sub-parts of a problem. This is mainly tested in question 4.5. Other related questions

include 3.1 (modifying HEX to produce TRIHEX) and earlier on, 1.1a, and 3.1. However, if we are arguing that Logo encourages learners to adopt a modular style, we would not expect them to exhibit such a style before they had encountered the idea of using procedures within procedures. For this reason, although in the analysis of errors there were five instances of lack of modularity early on (1.1, 1.2, 1.3, 3.1 and 3.2) they will not be looked at here. For those who got that far, additional question 1.2 also related to modularity and the un-numbered question: shapes. Question 4.5 is:

"Can you now construct a procedure FLOWER which has 6 petals and uses the PETAL procedure? Record all you attempts (and comments) on your comment sheets.

When you've successfully defined FLOWER, construct a procedure BORDER which gives you a border of flowers."

On this question three answers were marked as incorrect: two of these answers used the procedure PETAL but also used the primitive ARC, and the third didn't use PETAL. Their comments suggest that being able to call a procedure from another procedure by using its name is not understood:

"I had never thought I'd be able to do that (use the procedure name to call the procedure) it amazed me."

Additional question 1.2

This question was discussed in the previous section. The answers to it were also scored for the order in which the procedures were written and the degree of modularity. All scored highly. There is little evidence, altogether, from which to decide whether the subjects adopted a modular approach. However, it seems that the most competent and successful subjects did acquire this but many did not.

Recursion

Another claim often made about Logo is that students can learn about recursion through discovery, yet there is no strong empirical evidence for this, and some work (e.g. Kurland and Pea, 1985) suggests that students' models of recursion are usually faulty. There is some evidence that in mathematics, learning about iteration facilitates learning about recursion (Anzai and Uesato, 1982), and indeed this is the route usually chosen by curriculum developers. The Logo tutorial manual is no exception, and introduces recursion via iteration. This section looks at the models of recursion adopted by students in this study, and the problems they have.The data is discussed by first of all looking at students' answers to the in-text questions 5.1, 5.2, 5.3 and 5.4, and where available, additional questions 4.1, 4.2, and 5.1, and comments on the comment sheets and the interview transcripts.

Question 5.1

"You should be able to turn the simple iterative procedure:
BOX
REPEAT 4[FD 40 RT 90]
into a procedure Box that uses tail recursion.
Try it."

As we saw earlier, there were few problems with this question, and 8 of the 10

subjects who attempted it got it right. One error which did occur in one subject's early attempts was the non-agreement between variables in the title line and the body of the procedure, e.g:

```
RBOX
FD SID
RT ANG
RBOX SID ANG
```

It is not clear from this example, whether the confusion is connected with the fact that it is a recursive procedure. However, although this error was not common in this question it is a recurring problem that clearly hinders understanding of recursion. In this case, the person did go on to get the question right. Another problem concerns deciding which are the salient features (of recursion) in the example given. As with SOLO, subjects used the examples they were given as models to follow: for this to be successful subjects need to know which parts of the example program are crucial, and which are just specific to that example. In this question, one subject wrote the following program:

```
RBOX
FD 40 RT 90
FD 40 RT 90
```

and when she had seen the answer, she commented:

"I thought that the first and last line had to be some sort of an instruction as the example used the declaration which is a sort of an instruction."

The example she is referring to is:

```
RSTAR SID ANG
FD SID
RT ANG
RSTAR SID ANG
```

It is not clear, however, why having RBOX as the last line would not also be an instruction in terms of her definition. Subjects also had problems with question 5.2:

"Can you think of a way to modify the program for the BOX procedure that results in it drawing more and more boxes each half the size of the previous one?"
(BOX has been defined, and also RBOX).

Only half the subjects answered this correctly (5/10). Most of them tried to use SPIRAL, which had been given in the text, as a model (see Q5.2 in appendix 8.2). Of the six subjects who did not get the right answer, only three had made written attempts, and all three had the same problem which was identifying the role of INC in SPIRAL and working out what was needed in the new procedure for the side to be divided rather than to increase. Again this involves using the previous procedure as a plan; this time it is SPIRAL. The following bugs occurred in this question:

1. Not understanding that variable declaration cannot include an arithmetic function, e.g. that INC/2 is not acceptable in the following:

```
RBOX SID ANG INC/2
FD SID
RT ANG
RBOX SID+INC ANG INC
```

2 Not understanding the relationship between the INC in the SPIRAL example and what is required in the boxes example : i.e. not being able to use SPIRAL effectively as a plan.

E.g. S26's 3rd attempt is

RBOX SID ANG INC
FD SID
RT ANG
RBOX SID-INC/2 ANG INC

And he comments:

"INC will always give a spiral. I can't see any effect with the negative increase. Having looked at the answer I don't think question 5.2 was a very logical follow on from the SPIRAL example. I was concentrating on the INC command"

For this subject the problem is that there hasn't been an example of how '/2' is used, and he has attached it to INC, instead of to the appropriate variable, SID. The other two subjects had similar problems, and the extract transcript (8.2) below illustrates S17's attempt to use SPIRAL as a model, and her lack of understanding of the role of INC in SPIRAL. Her first and third attempts at RBOX are given in figures 8.3 and 8.4.

RBOX SID ANG INC
FD SID
RT ANG
RBOX SID+INC ANG INC

Figure 8.2: S17's first attempt at RBOX

RBOX SID ANG INC
REPEAT 4 [FD SID RT ANG]
RBOX SID+ANG INC
(with RBOX 100 90 10 - produced increasing square)

Figure 8.3: S17's fourth attempt at RBOX

1. "I had RBOX SID ANG INC, FD SID RT ANG RBOX SID + INC ANG INC; then I entered RBOX 300 90 and 300/2 and the computer said that it lacked information and then I tried bringing in the divide by 2 immediately after that, after the side length. I just put it over two there...and then..... I tried RBOX 300 90 and a half for the increase putting 1 over 2. I got an increasing square drawn, and it was increasing by 1, obviously just picking up the 1 so then I tried RBOX 300 and 90 side over 2 and was told that side had no meaning."

[E] " There's got to be a number in there"

2. "So then I tried..... it was dawning on me as well that they weren't nesting of course, I was spiralling,. in order to nest I had to complete a square and go from there which of course this other one didn't do. So I did RBOX side angle increase then I repeated 4 times the forward side rt angle and then went back to RBOX side plus increase angle increase and with RBOX 10 90 10 I produced an increasing square, ..."

Interview extract 8.2: S17 using SPIRAL as model for RBOX

Not understanding the use of '/' is also a major problem for this subject. It appears that she has identified the need for a divisor as she uses 1/2 in her second attempt (presumably the reasoning is that it might work analogously to multiplying by 1/2) and later uses 1/2 and /2 as values. Even though she has this problem, some of her attempts to solve the problem, and to deduce the relevant information are partly successful, e.g. she correctly predicts that the procedure must complete a square otherwise it will produce a spiral. It seems, however, that the combined problem of not understanding the use of'/' and not understanding the role of INC in SPIRAL make it an impossible problem for her to solve. Of course, in cases like this, the Logo style of making bugs and learning from them is not very effective. If a learner happens to have two (or more) bugs at once, and her conceptual grasp isn't very strong, it is difficult for her to make systematic changes to the program, observe the results and make inferences about the bug(s).

The third subject (S33) also had problems in using SPIRAL as an analog and in his

first attempt changed the INC of SPIRAL to DEC:

```
BOX SID ANG DEC
FD SID RT ANG
BOX SID1/2 ANG 1/2
```

This gave him the problem of where to use the divisor, so he ended up with two '1/2's on the last line. He compounded the problem by not typing in enough values on his first try. His final attempt was:

```
TO BOX
BOX SID ANG DEC
FD SID RT ANG
BOX SID+DEC ANG DEC
```

This procedure has a nice "near and yet so far" quality about it. Yet this procedure is identical to SPIRAL except that the INC is replaced by DEC. Compare SPIRAL below with BOX above:

```
TO SPIRAL
SPIRAL
SPIRAL SID ANG INC
FD SID RT ANG
SPIRAL SID+INC ANG INC
```

The problem would seem therefore to be in identifying INC as the crucial part that should be changed (which is correct) but not understanding its role sufficiently, nor the requirements of the new procedure, to be able to use the SPIRAL procedure as a model. Further evidence that INC is inappropriately transferred from SPIRAL and used is supported by S27's comments, where her first answer is:

```
RBOX NUM SID ANG INC
REPEAT NUM [FD SID RT ANG]
RBOX NUM SID/2 ANG INC
```

and comments:

"Realised that did not want INC as we are not increasing but dividing by two which was already in procedure. Took out INC from first and third lines"

Unlike most of the others, however, she is able, through successive modifications to achieve the correct result, and she commented:

"The final version was so simple compared to my first attempt. I think I am trying to cram every new command I have learnt into each procedure, instead of choosing the most suitable ones."

Question 5.3

This question follows the section on introducing the RECUR procedure and asks:

"What would happen if you typed in RECUR 200. Why?"

No-one could do this accurately. Some people made no attempt and two made the same prediction, e.g. S27:

"I think it would look like this:

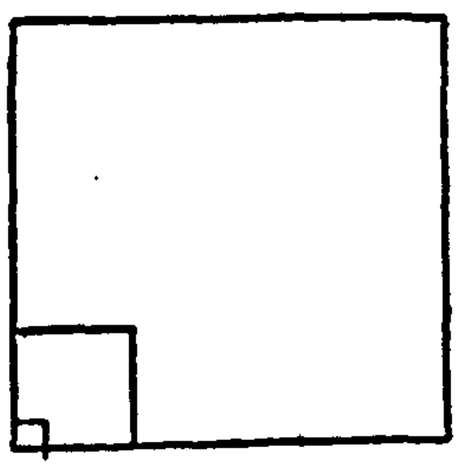


Figure 8.4: S27's answer to question 5.3

"After more thought I think I was reading the first line before the last line. This was the only way to make it do something. This did not seem to be what the program asked. I DO NOT THINK IT WILL WORK. .....Having seen RECUR 200 work and reading the procedure again I feel that I almost understand it but not quite."

What she found difficult to understand is explained in the interview. The problem is the order of execution: the fact that it starts with the smallest squares and then moves to the previous call to draw the squares a third bigger and so on. The salient parts of the interview are reproduced in interview extract 8.3 below: (The bracketed comments are those of the experimenter). The amended recur procedure is given first:

```
RECUR SID
REPEAT 4
IF SID < 20                          if SID is too small
[FD SID RT 90]                       go to next corner
[RECUR SID/3 FD SID RT 90]]          else do it smaller before moving on
```

1 "..I thought it would only draw one more corner, I couldn't see how.. it (would) do....four.

2 First I thought it would draw like that (see fig. 8.6) then I realised it wouldn't do anything because it was going to try and get smaller; then it .. asked what it would do then, - and I thought it would look like that (fig.8.7): I couldn't see how they managed to tell it to do one in each corner and move on the right distance for the next one. I watched it to try and work it out....

4 It .....has to get to its smallest point.....So, (for RECUR 50) I expect it to go straight down here and find.....the smallest one."

[E] "What will it do with repeat 4?"

6 "Once it's got to the smallest one it can go down to without stopping, it'll repeat ...on that corner....which I understand it'll do, it'll draw a square... .....but I can't see how it can go on to the next.....it'll go forward a side but why doesn't it go forward 50?"

[E] "If the side is too small it goes to the next corner otherwise it does it smaller before moving on?"

7 "Yeh. The RECUR SIDE/3 ..... is that giving you your figure to use in that one? Is that your 50 divided by 3 so that side will be....or will that still be 50?

8 When you watch it, it does a little one and then goes forward, so why doesn't it go forward 50? ..I watched it go around and it starts in a little corner, does the first square, moves that far, does the next one, and then it knows to move that far and does this and .. then works round....and I couldn't work out from reading this how it knew to go past that distance and then to move on to that distance and then that one.

9 It does all the small ones in that square, and then moves..... I couldn't see how side could change each side. The RECUR command hadn't been explained as such and I wondered whether the command RECUR did something else.....I could see how it would do the small ones first but I couldn't see how it could do a small one, go along and do another small one."

Interview extract 8.3: S27 talking about RECUR

This interview extract illustrates two points. She cannot predict the order of execution because she doesn't understand the flow of control: why execution

should happen in the order it does. In particular she cannot follow the backtracking in the example. This is the most prevailing problem and additional question Q X shows that no-one understood this. Secondly, she falls into the "demon" fallacy - see the section underlined in the above extract. This is the belief that the program contains some knowledge beyond what can be seen to be there. In this case, she believes that there may be a "command" RECUR - which is presumably like one of the primitives, and like FORWARD or LEFT contains some knowledge. This belief was also adopted by other subjects when other explanations evaded them.

The same problem is also evident in the transcript of another subject's interview. Her answer to 5.3 suggests that she sees a SQUARE plan in the RECUR procedure, - but again she cannot follow through the execution of the procedure. Other comments and transcripts tell the same story. Most people did not try to follow the RECUR procedure in the detail that S27 did, but it is clear that they couldn't follow its execution.

This analysis of students' problems in answering the questions on recursion discussed above indicates that their problems are in the following areas:

1. Declaration and use of variables
2. Passing values from procedure to procedure
3. Understanding the side effects of backtracking in total recursion
4. Extrapolating salient information from examples in the text
5. Understanding the commands available e.g. '/
6. Believing there's "extra knowledge" somehow available - as in "demon" example above

Given the problems with the tutorial manual, there is some evidence that the subjects' problems in using variables, procedures and recursion arise from the Logo curriculum, i.e. the tutorial manual. However, although the tutorial manual could be improved, novices encounter similar problems on other curricula (Kurland and Pea, op. cit). There is also evidence that subjects' problems with recursion are located in their own faulty models which are due in part to strong expectations that are not related to the current domain but the result of prior experience and everyday knowledge. Such examples were seen in SOLO (chapter 6) and the "demon" example given earlier here is another such example.

Other studies have similar examples. Kurland and Pea (op. cit) discuss bugs such as "assignment of intentionality to program code", "treating programs as conversation like", "overgeneralisation of natural language semantics". What these have in common, according to Pea, is that they are based on a predominant analogy that guides students' behaviour - which is conversing with a human. Of course, such an analogy is problematic. In this study, however, this does not account sufficiently well for the problems we have seen.

It is also the case that recursion is an inherently difficult concept. There is, however, evidence that "self- guided" discovery (Papert-style) is not effective (Kurland and Pea, op. cit.). There is also evidence that with quite different forms of instruction, and a different language, recursion is problematic (Kahney, 1982).

There are enough examples of student' problems with recursion with different curricula and languages that it is clear that getting the "formula" right for its instruction is no trivial matter. But at the same time, there is no evidence that it is an intractable problem, nor that the majority of problems we have seen are

resistant to careful instructional design.

List processing

Lists are introduced as a type of variable. Tail recursion had been covered, and also the command MAKE in section 5.4 "Fixing the values of variables". Section 5.5 is headed "Types of variables: the commands WORDQ, NUMBERQ, LISTQ".

There are no in-text questions on list processing so the evidence is from the comments and the interviews. First of all however, it should be remembered that there were errata sheets on this section, and if students failed to notice these, the WANDER procedure would not work. They therefore could not continue on to CYCLE. The evidence from the comment sheets and interviews suggest that the section on lists was not understood. Two subjects commented that more examples were needed on this section.

Altogether there were ten students still active at this point as three had given up. These ten fall into three categories: those who were successful (3), those with limited success (3), and those who had no success at all (4). The first group were able to get the example procedures given in this section working properly, but they did have problems to start with. In the three categories, the comments almost speak for themselves. The interview extract below is of S31 who was successful:

1 "Having no line numbers makes it impossible, when checking a procedure, to know whether or not you pressed RETURN at the end of a line, or put in spaces (later edited out) or used the cursor control keys (as I did in WANDER).

2 After typing in the procedure WANDER on p 70 and WANDER [10 20 30] [45 90 135] I get an error message 'Not enough input in procedure WANDER' Can't see why?

3 (Later he finds out)"...When I typed in WANDER and it stopped with the error message, as it's not a question, I think the first two lines of page 70 should follow the diagram on page 69, otherwise you might, as I did, think that I'd done something wrong"

Interview extract 8.4: S31 talking about using WANDER

The problem is that the example given, as it stands, is meant to end with an error message, but of course this is not explicit! The text states:

"Prepare for drawing and type the instruction WANDER [10 20 10 20 10] [90 135 135 90 45]. When I tried it the turtle left this track as it wandered." Over the page, the text continues: "It halted with an error message when it had used up all the numbers in the lists and could not find the first item of the empty list [ ]!"

It is not helpful to give instructions to the student which will lead to an error message, - also it is unlikely that any student will be able to follow the execution sufficiently to realise that it "could not find the first item of an empty list [ ]"

The evidence from the comment sheets and interviews, therefore, is that the section on lists was not understood. Rather than helping, the tutorial manual obstructed the subjects' chances of learning about lists. There are a number of problems, which together make this section very difficult. First of all there are the mistakes in the text, and although errata sheets were given, subjects often either missed them, or didn't consult them at the appropriate time. Using REST instead

of its replacement BUTFIRST accounts for quite a lot of the problems. The second problem is when carriage returns end up in the wrong place: as one subject said, it is hard to determine where they are, so hard to correct, even if you know where they should be!

The first example of WANDER given in the text ends (untidily as the text later points out) with an error message of "NOT ENOUGH INPUT". This is meant to happen, and is discussed in the text. However, the reader is not warned in advance, and of course, as one subject pointed out (interview extract 8.4 above) it is not clear to subjects, especially after all the previous problems, that this is a deliberate error. Finally, the procedure WANDER is both uninteresting and hard to follow. It is not surprising, then, that this section on lists was rather unsuccessful.

8.6 DISCUSSION AND CONCLUSIONS

Difficulties

Parts of the curriculum were very hard: recursion, list processing, functions. Students had not got sufficient understanding of the pre-requisites needed for these sections. There were not enough examples for illustrating and introducing new ideas, and some of the examples that were used were badly chosen. In particular, the number of mathematical examples (mostly in sections 6 and 7) led to two problems. First of all, it is not appropriate to many learners' interests and therefore not motivating. Secondly, the examples did not match the subjects' capabilities. The right angle triangle example for instance, demanded knowledge which the students hadn't got, - and in any case they could not see the point of the exercise. It is also possible that there is a stronger effect than this which is that

many people have negative associations with mathematics, and may therefore find such examples off-putting before they start. The problems that subjects had, with variables, modularity, recursion and list-processing, are all problems which have been reported in the literature, but this particular curriculum exacerbated these problems.

More specific problems were consistent in the subjects' reports and the evaluators' comments, such as the ambiguities and inaccuracies that appear in the text; that some commands are not explained clearly enough; that learners are invited to make errors. Examples of these have been given, and the point made that they are more problematic for the novice at a distance than for more experienced learners in a face to face situation. The error messages, in particular, were not helpful, because they were not related to the underlying causes of the errors, nor to any model that subjects may have been developed. The subjects had more syntactical problems than would have been expected, even taking into account the unhelpful error messages. many of these problems can be traced to the tutorial manual, and were not, therefore, predicted by the analysis of the conceptual model in chapter 4.

Subjects had problems in abstracting salient information from examples given in the text. Partly because of errors in the text, they failed (or failed at first) to get the example programs working - and received error messages which were singularly unhelpful in explaining the problem at an appropriate level. Their specific problems concerned using and understanding variables - especially passing values from procedure to procedure, faulty models of recursion and list processing. Interestingly, there is little evidence of the looping model of recursion which Kahney (1982) found and is in evidence in SOLO. In the SOLO manual,

students are "talked through" flow of control, and also encouraged to follow through the execution of the example recursive procedure, and to notice which triples are added into the database. In the Logo manual, students are invited to turn an iterative procedure into a recursive procedure, and subsequently to construct a new recursive procedure to draw a figure where the length of the side increases each time the figure is drawn. They are not guided through the execution in any detail, and in fact students could not follow through the execution of completely recursive procedures.

Unlike SOLO, the Logo manual introduces interation first and then recursion. This is a common strategy; but when Weidenbeck (1989) conducted some experiments on learning to computer recursive mathematical functions, she found that prior work with iterative examples did not facilitate the subsequent learning of recursive procedures. Although the subjects' lack of success in list processing is related to their problems in understanding recursion, other attempts to teach list processing to children and adults have not been successful (Sharples, 1985). Sharples argues that the conceptual model for turtle geometry does not easily extend to list processing, as instead of a model of issuing commands to an object, a different model is required, of functions that take arguments.

A programming language or educational philosophy

The problems discussed above need to be viewed in the context of Logo's educational philosophy. Some of the teaching techniques used in the OL tutorial manual are inappropriate for distance learning. One commonly used teaching technique, especially in problem solving, is to lead or encourage students down a particular path which leads to what seems like an impasse, at which point the

teacher introduces a method for resolving the problem. This is a well known teaching strategy, which could be called the "garden path" technique. It is used in a couple of places in the OL manual: for example where the user is invited to try out a recursive procedure which turns out to be infinitely recursive, as a way of introducing stop conditions. However, such a practice is dangerous in distance learning, as the author doesn't know how long the student will stop and worry, before finding out that it was a delibarate mistake! This certainly happened to the subjects learning Logo, in the list processing section for example.

The difficulties experienced by the students led to problems with the exploratory, "diagnose your bugs" style. Where there is more than one bug, and these may be interacting, and the students' processing models are faulty, they are unlikely, themselves, to diagnose the problems. Certainly there were few instances where subjects made comments such as: "Aha, now I realise what my mistake was!" This is related to the issue of the two approaches to learning and teaching Logo which was raised in the introduction. The aim of the OL tutorial manual is not clear but appears to be an attempt to take both approaches simultaneously: for learners to both learn Logo-as-a-programming language and the Logo educational philosophy. Learners need to have grasped non-trivial programming fundamentals in order to cope with the latter part of the curriculum. The author's style and tone is evidence of the second approach.

Yet although the intention is for students to learn to program in Logo, one of the effects of the learner centred approach adopted here is that programming is not taught. In chapters 6 and 7, when discussing SOLO and PT501, it was argued that the stages of programming were not taught in an explicit way: too much work was left to the learner in terms of deducing the relevant attributes in the examples.

Here, programming is not taught because the learner is supposed to discover it! Unfortunately, however, the combination of an adult learner and this manual does not provide the requisite conditions where this can happen! Teaching through discovery learning needs a very carefully worked out approach which the manual doesn't have. For example, it is not sufficient to mention, once, that bugs are nice likeable beasts for diagnosing errors in our thinking. Most adult learners (at whom this tutorial is aimed), have a history of regarding errors as failures, which are to be avoided at all costs. They are not used to the amount of time, uncertainty and frustration that a discovery learning approach involves, and it is likely that much of their learning experiences are of a didactic approach. In fact Tennyson and Rasch (1988) give Logo as an example of a domain which is eminently suited for what they call "self directed experiences", but add the caution that, because of the time necessary for participating in the creative activities associated with self directed experiences, educators should ensure that they provide sufficient learning time. In the case of Open Logo, the subjects did comment that they felt that they had not had enough examples, which is indicative of the problems of trying to take this kind of approach but using a limited amount of print. In the end, the manual fails to either teach Logo or to convince students about the Logo philosophy.

On the positive side however, the excitement of making the abstract, concrete and therefore one's own for exploration, did not get totally lost. This wasn't the whole Logo experience, but earlier on (before it got too hard) students commented on the excitement of discovery and making connections, and in some cases making connections for the first time. The chapter finishes with an interview with S33 who is explaining how he struggled to find out what commands were needed to draw a circle:

"It was at this point I think that I had to really start thinking about angles, and I found it quite

exciting, ...because I couldn't get it right - I suppose if I'd just got it right it would have been boring..... I found the circle very difficult,.... and I couldn't get to sleep for three hours after that, - I went through all of primary school geometry. I was sweating with this. I knew something had to balance between the number outside the brackets and the number inside the brackets because I knew that there were 360 degrees in a circle,..... but it took me a long time to work out what the things that I had to relate were, and which were the ones that didn't matter too much. And I found that quite exciting, but it took me a long time."

Chapter 9

Learning to use DESMOND

Contents

9.1 INTRODUCTION

Chapter 7 discussed the problems which learners have in learning the PT501 assembly language. It was suggested that learners' problems can be viewed as instructional, arising from their interpretations of the text: i.e. 1) they develop mental models which are often inaccurate, and 2) the low level at which the students are working is not conducive to the development of plans. However, it is not clear to what extent these results can be attributed to the conceptual model being less effective than hoped, and the fact that the clarity of the instructional text could have been improved, and subjects had insufficient guidance on interpreting what they saw on the display. These problems should be overcome by using a different microcomputer system (DESMOND) and instructional material which has benefitted from our experiences of students using the 8049 assembler (course code PT501) and its successor, PT502.

This chapter reports on students' experiences with DESMOND as part of the study of the transfer of skills between high and low level programming languages which was discussed in the last chapter. These studies have the same aim as the earlier studies reported in chapters 6 and 7: of investigating learning a high level language followed by a low level language and vice versa, concentrating on the transfer of low level skills. LOGO was the high level language studied, and the low level language was incorporated in a small hand held microcomputer called DESMOND. Students learnt a little machine code, but mainly assembler. There was no strong evidence of transfer, and this result is discussed here (and in chapter 8 which also dealt with subjects' experiences with LOGO). Because of this result, this chapter reports mainly on subjects' experiences with DESMOND.

9.2 THE STUDY

Design

The design of the study is the same as for Logo. Twenty people began the study, and were allocated to two groups. Group 1 worked on DESMOND first followed by LOGO whilst group 2 worked on LOGO and then DESMOND. Most of the subjects were female: with one man in group 1, and two men in group 2. The number of subjects was limited by hardware availability. Six people from group 1 completed half or more of the curriculum, and 7 from group 2. There was, however, steady attrition as shown in table 9.1 below, and by the end of the course only two people remained in group 1 and 3 in group 2. The main points of attrition are at the end of chapters 1, 2, 4 and 5. The number of subjects, therefore, that the analysis is based on varies according to the DESMOND chapter in question. This is again shown in table 9.1.

Subjects

All the subjects who took part in this study (except one) were Open University employees. They were recruited via an advertisement in the University's in-house magazine and were told that to be eligible they should have no programming experience. Information was collected about their academic qualifications, but they were not selected on this basis.

Chapter	Total active	Gp 1	Gp 2	Total	Losses (by chapter)	
		D->L	L->D		Gp 1	Gp 2
1	20	10	10	4	3	1
2	16	7	9	3	1	2
3	13	6	7	0	0	0
4	13	6	7	3	1	2
5	10	5	5	3	1	2
6	7	4	3	2	2	0
7	5	2	3	-	-	-

Table 9.1: Attrition by group and chapter

Task

Subjects were given a DESMOND practical book and a DESMOND computer and asked to work through the practical book, to attempt all exercises and problems, and to fill in comment sheets for each of the 7 chapters. An estimated time for completing each curriculum was given as 20 hours. Examples of the in-text exercises and answers and comment sheets are given in appendices 9.1 and 9.2.

Interviews were set up every two weeks for pairs of students in order to discuss the the work that they had done the previous two weeks and particularly any problems they were having. The in-text exercises were also discussed at these interview sessions and subjects were asked additional questions. The interview sessions were not always held at the planned intervals: mainly because people had got stuck, and therefore had not completed what they intended to. In these cases either the interview was delayed, or more commonly, the experimenter and subjects met to discuss the problems they were having and sort them out. Subjects were told that they could phone up the experimenter at any time if they had a problem, but that she would try not to give them the answer but would help and encourage them to work out the answer for themselves. The answers to the exercises and problems were taken out of the booklet which was given to subjects but they were available at the interview sessions to be consulted. In the later chapters, or if people were struggling, they were given copies of the answers but asked to try to do the

exercises and problems without referring to them.

9.3 RESULTS

The chapters vary considerably in terms of how many subjects successfully completed the exercises. Chapters 2 and 4 in particular were perceived as difficult, and more subjects attempted the exercises in chapter 3 (which were not programming exercises). Chapter 5 was also difficult, but after this point, the small number of students who stayed the course and attempted the exercises in chapters 6 and 7 had quite a good success rate. Table 9.2 below gives the percentage scored by each group for all the exercises in each chapter, and also the percentage of exercises attempted.

	Group 1		**Group 2**	
Chapter	% Attempted	% Correct	% Attempted	% Correct
1	79.2	81.4	84.8	92.8
2	56.9	66.7	78.0	87.5
3	93.6	60	92.0	50
4	43.3	70	47.6	88.3
5	38.6	40.9	54.5	43.6
6	54.5	81.8	75.8	100
7	20.8	100	63.9	89.2

Table 9.2: Percentage of exercises attempted and scored correct in each chapter averaged across each group

It can be seen that group 2 (who are learning DESMOND after LOGO) perform better than group 1 on most of the exercises: the exceptions being those in chapters 3 and 7. The difference is largest for chapters 2, 5 and 6. This difference was tested using a Mann-Whitney test and was significant for chapter 6, but not for the other chapters. (See tables 9A.1 - 9A.6 in appendix 9.4). Although the scores for chapters 3 and 7 would appear to be against this trend, this is partly explained for

chapter 3 by the fact that chapter 3 contained very few exercises and no programming exercises and many subjects omitted them. One noticeable difference between the two groups is both in attrition and the number of exercises completed. More subjects in group two completed the course, and they attempted more questions.

Basing the subjects' percentage scores for each chapter on the total number of exercises rather than the total number attempted therefore gives group two an advantage, but it seems reasonable to suppose that the exercises which subjects did not attempt were likely to be those they could not do, and so using the total number of attempts would disadvantage the subjects who attempted (as requested) to complete all of the exercises.

It is not clear from the overall results then, whether any slight advantage gained by group 2 is due to staying power, attempting more exercises, or a higher level of programming competence. The rest of this section gives a categorisation of errors made in the programming tasks, and the following two sections look at the categorisation of problems, and considers whether the two groups differ in the number and types of problems experienced.

Categorisation of errors

Some of the exercises set were programming tasks. Of these, 61 programs which were marked as incorrect were analysed for errors. Two points need to be made here. 1) Some of these programs did run successfully in that they produced the desired effect, but contained bugs and were therefore marked incorrect (e.g. a common mistake is an unneeded jump from the end of the program back to the beginning). 2) The number of programs marked as incorrect is not the same as the number of programming tasks which were not successfully completed. Many of the unsuccessful attempts produced very incomplete programs or a failure to write

anything down at all: although at the interviews it was clear that even where nothing was written there had often been several attempts - perhaps resulting only in jottings on scrap paper. Most of these, however, had not been preserved, and therefore such attempts are not open to analysis, but are indicative of the difficulties which people had. Table 9.3 below gives a breakdown of the errors by chapter and group.

Chapter	No. of programming tasks analysed	No of errors		
		Gp1	Gp2	Total
2	14	8	7	15
4	21	8	15	23
5	11	4	23	27
6	9	13	5	18
7	6	4	3	7
Total:	61	37	52	89

Table 9.3: Errors by chapter

The number of errors as a proportion to the number of programs increases in chapters 5 and 6. This partly reflects a difference in the types of problems encountered in the different chapters, and the fact that the programs in later chapters were more difficult. In chapter 4, unsuccessful programs were often almost total failures, consisting of hardly any code, or no attempt, whilst in chapters 5 and 6, good attempts were made, although these often contained more than one error. This relationship doesn't hold in chapter 7 however, where on the whole the exercises are less demanding. There are more errors among group 2 subjects than group 1, but this difference is not significant, and is not related to the group's overall performance as there were often a number of errors in a single program. The errors were categorised into different types, which are given in table 9.4, along with frequency of occurrence.

		Total		Gp 1	Gp 2
1	Flow of control	37	42%	13.8%	27.6%
2	Missing/incomplete code	13	15%	6.9%	8.0%
3	Missing plan or inappropriate plan	10	12%	4.6%	6.9%
4	Unnecessary code	8	9%	1.1%	6.9%
5	Wrong instruction	5	6%	2.3%	3.5%
6	Factual error	3	4%	2.3%	1.1%
7	Incorrect code	10	12%	8.0%	3.5%
8	Other "slips"	3	3.4%	3.5%	3.5%

(Total no. of errors = 89]

Table 9.4: Categories and frequency of errors

In table 9.4 the errors from the two groups have been allocated into 8 different categories. The biggest differences are in groups 1 and 4. In the first group, flow of control, there are twice as many errors made by group 2 subjects, but this difference is not significant (see table 9A.12 in appendix 9.4). Most of the difference is in chapter 5 where the exercises give the most scope for such errors. The general picture of the two groups, then, is that group 2 is more successful in carrying out the exercises, but also makes the most errors, especially flow of control errors. As there are more group 2 subjects left by the end of the course this suggests that they are more tenacious: both in their attempts to carry out the exercises and in staying the course. The eight different categories of errors are defined and discussed briefly below and a fuller treatment of the problems

experienced by subjects, many of which are indicated by these errors, is given in section 9.4.

1 Flow of control

Flow of control errors occurred throughout chapters 2 to 7, but the type of error changed as the programming tasks became more complex. For example exercise 4.3 requires a very short answer with little room for individual variation. The following code gives the standard solution:

```
00 LDA  91;  Load the value in address 91 into the accumulator
02 ADI 001;  Add 1
04 STA  91:  Store the result in 91
```

Each of the answers marked as incorrect 'worked'; i.e. they achieved the desired result, but contained an additional line with a jump back to the beginning of the program: 06 JMP 000. In later exercises, this particular problem was less common. The other errors which were classed as flow of control included control "falling through". This error also occurred in more complex programs where it would be harder to diagnose. (Such errors were rarely diagnosed by the subjects themselves). Other flow of control errors include unreachable code, jumping back to the wrong place, jumping to the next line, and so many jumps that it was hard to diagnose what the intention of the programmer was.

2 Missing code

The next largest category was missing or incomplete code and this was usually in the form of an unfinished program, - unfinished because the person didn't know

how to go about it. There were often comments such as "I just didn't know how to do this part" or "I didn't know where to start". Less common examples categorised as missing/incomplete code consisted of only one line missing.

3 Missing or inappropriate plan

Plans are usually two or more instructions used together to achieve a certain result, - for example displaying the state of a device, or "counting"[1]. The errors in this category were all to do with failures to use plans correctly. For example exercise 5.9 is to write a program for which it is necessary to use the count plan (discussed in appendix 9.3). In one answer the count section contained an initial clear, i.e. the first instruction set it to zero, so that it was only ever able to count as far as one. This initialisation should have occurred outside the count routine. This was classified as a count error, although it could arguably have been a flow of control error. The reasoning was that if the count plan had been properly understood it would have included the notion of initialisation outside the loop.

Another program had missing code - because the person completing the exercise didn't know where to put the "count". This was also classified as a plan error, for the same reason as above, as was the program with large chunks of missing code and the comment: "I can't work out how to keep a record of the count and in what position".

4 Unnecessary code

This was encountered throughout chapters 2 - 6, but was more common at the beginning. It consists of extra, unneeded code, for example, including a binary as well as a denary display routine when only one is asked for. Unnecessary jumps

[1]But see plan 5 :N JMP N" (given in appendix 7.5)

back to the beginning, however, are not included here as they are included under flow of control.

5 Wrong instruction

Examples of this include using STA instead of JSR; JSR instead of JMP, and ASCII display instead of denary display.

6 Factual error

An example of this would be using the wrong binary or denary number to light a particular lamp, as opposed to using the wrong instruction (as in 5 above).

7 Incorrect code

This covers instances where the code produced has little resemblance to the correct code, and it cannot be classified under 1 - 6 above. There are only 4 instances of this and they include programs which are such a long way from the correct version and so confused that it is hard to analyse them.

8 Other "slips"

There are only 3 instances of this and I have called them "slips" as they appear to be just that: not saving a mask (1), using the wrong mask (1), and not restoring the value of the keyboard after decreasing it (1).

9.4 THE DIFFERENT GROUPS OF PROBLEMS

Section 9.5 analyses the errors along with other data (comments and interviews) to look at the kinds of problems experienced by subjects. In order to do this it is necessary to categorise the problems into three main groups. This section outlines the different groups. These are:

1. Programming, i.e. related to the domain itself.

Within this category the subjects were engaged in two different types of tasks: programming and non-programming. The programming tasks are those exercises that consist of writing programs. Non-programming tasks are reading the text, trying to understand it, operating DESMOND and commenting, etc. All the problems in this group are related to the domain itself, programming, as opposed to the teaching style or strategies used which is included in the next group, or affective problems such as lack of confidence which are in the third group.

2. Pedagogy

Problems to do with the style or method of instruction. This group includes problems such as jumps in the level of difficulty.

3. Affective

This groups covers affective problems such as subjects' attitudes to learning to program, and lack of confidence.

The different groups of problems are shown in table 9.5 below:

	Programming	Pedagogy	Affective
Group	1	2	3
Problem	1 Flow of control 2 Plan related 3 Wrong instruction 4 Terms or concepts 5 Instructions 6 Operations 7 Syntactic	8 Conceptual level 9 Task 10 Level of understanding	11 Learning to program 12 lack of confidence 13 Other

Table 9.5: Grouping of the different problems

The next section outlines the problems in the different groups, and the relationship between them, and then they are discussed in detail in section 9.5.

Outline of different problem groups

Group 1: Programming

The first three problems in group 1 are:

1. *Flow of control problems*
2. *Missing or inappropriate plan*
3. *Using the wrong instruction*

These domain related problems which form the bulk of those discussed include errors 1, 3 and 4 from table 9.4 on page 9. Error no. 2, missing or incomplete code, which accounted for 15% of the errors is not considered here under problems as it is symptomatic: an indication of other problems rather than a problem in itself. Data from the comment sheets and interview transcripts was looked at to try to determine the reasons for not being able to write the code.

Although flow of control errors make up the largest proportion of errors (42%) as opposed to 12% for plan errors (and incorrect code), it is necessary to pay at least equal attention to the plan problems. This is because in some instances the plan

problem may be primary, and may therefore lead to the flow of control problems: (in fact it was sometimes difficult to decide in which category an error belonged), and also because analysis of the instructional text revealed a weakness in both these areas. Unnecessary code (4 in table 9.4) is not examined in any further detail here. This is because the other data does not suggest that it is a significant problem. The same is largely true for the remaining 3 types of errors, although factual errors and incorrect code are discussed.

The next four problems are concerned with non- programming tasks, although naturally some of these affected programming tasks also. These were :

4 Confusion between two terms or concepts

The main example of this which will be discussed in detail is the address/memory location distinction.

5 Confusion between two operations and problems understanding how a particular operation works

The example of this problem which will be discussed in detail is that of memory mapped routines.

6 *Syntactic errors*

These are included among non-programming tasks because they mainly occurred whilst exploring DESMOND rather than programming. The most interesting example is related to DESMOND's mental model, - and it is this which will be discussed.

Group 2: Instructional

This second group includes statements or beliefs about the pedagogy, and need to be considered alongside the main problems about plans and flow of control, as they are to do with how these areas are taught - or not taught. They are mainly supported by qualitative data: what was said at the interviews and written on the comment sheets.

6 *Jump in conceptual level*

This problem is concerned with the level of difficulty changing rapidly, or the feeling that exercises required a leap of understanding from what had gone before.

7 *Expectations about the task*

8 *Expectations about the level of understanding required*

This represents comments and expressions that the activity or concept being studied has not been understood on a deep enough level, even though, if it is an activity, it may have been successfully carried out.

Group 3: Affective

This final group is more affective in nature, and includes comments about the activities, feelings of lack of confidence and so on. These include:

10 Expectations about learning to program

11 Lack of confidence

Finally there is a 'catch-all' category which lies outside this grouping, headed simply:

12 Other problems

Each of sections 5 - 8 will be concerned with one of the four groups and will discuss examples of each of the problems, within the three groups. Some types of problems are more interesting or more important than others and this is reflected in the number of examples given, and the amount of discussion for each.

9.5 GROUP 1: PROGRAMMING

Flow of control

This includes all problems and errors which are to do with flow-of control including unnecessary repeats (jumps to the start of a program), getting the sequence right (doing things in the right order), problems in testing conditions, using subroutines and also comments about flow of control. The evidence of

problems early on in learning DESMOND comes from the comments, as the problems in writing programs manifest themselves later when more programs are being written. Here is a comment from S17[2]:

"Sometimes the word go is used and sometimes there's jump to and sometimes there's jump back...and I don't know whether jump and go and jump back or jump to or go back are really all the same way of saying the same thing - I know jump the way it has been used here always seems to lead to repeat but does it always? "

One problem that occurs quite early on is whether to insert the instruction JMP 000 at the end of a program to repeat it. This instruction, and this form of its use is introduced very early on. The very first program given is one which displays the state of the switches:

```
LDA 104
JSR 205
JMP 000
```

The JMP 000 is needed to repeat the program in order to keep checking the state of the switches - which may have been changed. This point is explained, but what isn't explained or given is a number of examples where JMP 000 is needed and where it is not.

Clearly, for programs to work, the instructions need to be in the right sequence.

---

[2]Each subject taking part in these studies has been allocated a different number. The subjects in the DESMOND/LOGO studies reported in this chapter and chapter 8 are S16 - S36

This was also a recurring problem. One problem was control "falling through": to some point of the program that should never be reached under those conditions. For example in problem 4.8, students are asked to write a program that will take a value stored at an address and light one of the four lamps according to whether the value is 1, 2, 3 or 4. Only one subject completed this program correctly. There were 9 subjects active at this point and 4 attempts. Below is an attempt by S28 to do exercise 4.8 which illustrates the problem of control "falling through":

```
00  LDA 91          ; Load into accumulator number from
                      location 91
02  DEC             ; Decrease by 1
04  JZ  20          ; Value now 0, therefore was 1
                      Jump to 20
06  DEC             ; Decrease by 1
08  JZ  30          ; Was 2
                      Jump to 30
10  DEC             : Decrease by 1
12  JZ  40          ; Was 3
                      Jump to 40
14  DEC             ; Decrease by 1
16  JZ  50          ; Was 4
                      Jump to 50
20  LDI 001         ; Put 1 in the accumulator
22  STA 101         ; Send to lamps (to light lamp 1)
24  JMP 000         ; Jump back to beginning
30  LDI 002         ; Put 2 in the accumulator
32  STA 101         ; Light lamp 2
34  JMP 000
40  LDI 004         ; Put 4 in accumulator
42  STA 101         ; Light lamp 3
44  JMP 000         ;
50  LDI 008         ; Put 8 in accumulator
52  STA 101         ; Light lamp 4
54  JMP 000
```

This program was written after an interview where it was explained how DEC and JZ could be combined so as to test whether the contents of the accumulator were 1,

2, 3 or 4. It works where 1, 2, 3 or 4 is in 91 and lights the correct lamps. However, if any number higher that 4 is input it falls through to line 20 (having decremented 4 times) and lights the red lamp. S6 was aware of this problem - but could not diagnose the bug. Correcting the bug is quite simple - a stop condition could be inserted after all the tests for the contents of the accumulator e.g 18 JMP 18 would do. Exercises 4.9, 4.11 and 4.12 were all difficult, and had flow of control errors. For example, in 4.11 the sequencing problem is compounded by the problem of where to store the count:

S27 said:

"Am having difficulty thinking of where in program to store the count. Logically it should be between the turning the lamp on point and the key pressed part but I can't see how to put it there without overwriting the address .....Looking back at this exercise I didn't realise only key '1' was being used."

Subroutines

Subroutines are introduced in chapter 5. The text gives an optional section which discusses return addresses and the stack. The only person (S10) who commented that she had read this, said she found it confusing. In exercise 5.2, subroutines were used incorrectly, and in 5.3 subroutines were not used when needed and when they were used were used again incorrectly.

Testing conditions

Problems to do with testing for certain conditions are hard to separate from plan related problems in that if a plan is well understood it will include the associated test conditions. Plans such as those for testing the keyboard, the delay loop, testing for numbers other than zero and for counting routines, are commonly used and needed. These were often not acquired. For example, the test for a number

requires decrementing by zero and using the number of decrements to determine the number. This combining of DEC and JZ was not given: most people had great difficulty in working it out, and many failed to do so. They then had problems in carrying out the following exercises, all of which demanded this routine. Some people found the task of writing programs so daunting that they failed to include test conditions altogether. For example in 4.9, one person couldn't get out of the counting routine, - and it turned out that she didn't have a test! Similarly in chapter 5, failure to include a test accounted for problems in exercise 5.9.

Missing or inappropriate plan

Examples of plans in DESMOND

Plans are two or more instructions used together to achieve a certain result. They are also generic and can be used in various different situations. This is illustrated in the first example below. which has a slot for "address of device". This generic property is not made explicit in DESMOND.

All the plans included in the DESMOND curriculum are given in appendix 9.3, which also indicates whether a plan is given in the text, or needs to be worked out by the student, and how they are combined in the exercises and problems. Below are two examples of plans which are typical. The first example is the very first plan encountered in DESMOND.

Plan	Function	Outline
1	Display the state of a device	1 LDA [Address of device]
1b(Sub plan)	Display routine	2 JSR 200 or 205
1c(Sub plan)	Loop	3 JMP 000 [Repeat to update display]

Note that there are two sub-plans - display routine, (1b) and loop (1c). The address in (1) could be 104, 106, 107 or 108, and either display routine could be used.

This plan is used in the very first program given in activity 1D on page 25 which is:

```
00 LDA  104
02 JSR   205
04 JMP  000
```

This program uses the state of switches routine at location 104 to inspect the 8 switches. In further activities different devices are displayed by changing the contents of address 01 (now containing 104) to 108 (angle sensor), 107 (light sensor) and 106 (heat sensor). This plan is therefore given and explored as part of the programs which display the state of the devices (as above) and is analysed in chapter 2 where the rationale for continually repeating the program to update the display is also explained. The notion of abstracting a plan for displaying the state of a device from the programs for displaying particular devices is not, however, made explicit. This is true for all the plans.

The second example is plan no 7, the COUNT plan:

Plan	Function	Outline
7	Count	1 LDA N1
		2 ADI N2
		3 STA LOC1

This plan is not given in this form. The ADI instruction is introduced at at the same time as ADD, and exercise 4.2 involves typing in a program from chapter 1 (in numerical code) and viewing it in assembly mode. The program is:

```
ADI     001
STA     101
```

The count plan (7 above) needs to be worked out for exercise 4.3:

"Write a program that will take the value stored at memory location with address 91, increase it by 1 and store the answer back at the same location."

The answer:

```
LDA     91
ADI     01
STA     91
```

is plan 7 above, and a little further on its function is mentioned:

"In longer programs it can be necessary to keep count of how many times a particular thing has happened. The program above indicates how this can be done"

Given the problems that were encountered with the count plan, introducing it explicitly and giving further examples here would have been helpful.

Problems relating to the use of plans

This section analyses some of the exercises and problems in chapters 4 and 5 further by using this plan classification. This analysis combines the quantitative and the qualitative data.

Exercises which were problematic can often be seen in terms of failures to recognise and apply such plans. Chapter 4 contains 9 exercises involving programming. 40 incorrect programs (or attempts at programs) were produced among the responses to these 9 exercises. None of the incorrect answers to exercises 4.3 and 4.4 contained plan errors.

Exercise 4.5

Exercise 4.5 is classed as a problem, which means that it is considered harder than the exercises. The task is to write a program to multiply the value at location 91 by 16 and store the answer in location 92.

Eight subjects completed this exercise successfully (from a total of 13). Of the remaining 5, two made no attempt, and two made very incomplete attempts, where the final programs they produced did not resemble the solution at all.

The comments, however, are more revealing about the kinds of problems people had. One problem here is that the text misleads the reader as S17 comments:

"the line in the question 'Do not add the number to itself 15 times, try to think of a neater method' is completely misleading, since the solution given is only a variation and not a completely different method as the question seems to be suggesting. It would be a very academic point to say that 2 + 2 is different from double two."

and this misinterpretation by the subject (that she should be looking for a different method rather than a variation) prevents the activation of the appropriate plan, which is the multiply plan:

LDA

ADD

STA.

Exercise 4.7

This exercise involves working out a program to determine the value stored at an address, and if it is one, lighting a lamp. Four subjects made attempts that could be analysed: of these, 2 were correct, one contained one error and one contained two errors. The remaining subjects made no attempt or made an attempt that was too incomplete for analysis. The 3 errors included 2 flow-of-control errors and one missing code. On talking to the subjects and reading their comments it was clear that they did not know how to test for the value 1, which is not surprising as this plan has never been given to them. The answer involves combining DEC and JZ as we saw (**plan 11**). The extract below is from one interview transcript (S31):

"Um, well I was stuck because I couldn't seem to work out how I could find out if it was 1 stored in that, I wasn't sure quite how to do that, and I did have a program that when I put a number it went through a loop and decreased it until it was 1 and then when it was 1 it lit the yellow lamp, but obviously that's not exactly what's wanted, so I just got that far and left it at that, because I felt, you know, I couldn't really work out how to find out how it was 1 or not. That was the main problem with I think a lot of the others as well, the fact that you had to be able to recognise that there was a certain value stored in the accumulator, and that was the main problem I feel with all of them."

Another said:

"..it was just a complete blank. I read through again thinking I must have missed something, it hasn't sunk in,..... I must admit I hadn't gone back to it. I think it's the if one, light the yellow lamp, was the problem....I mean if it was zero, fair enough, but I couldn't see how you could do it, not with the instructions we've got so far. You could jump if it's zero, or have an unconditional jump, mmmm.... I really couldn't see how to do this one at all, I didn't do the problems either, 4.8." (S24)

Problem 4.8

This involved writing a program to take a value stored at an address and lighting each of 4 lamps according to whether the value is 1, 2, 3 or 4. Again this caused a lot of problems. One person managed to do it eventually, - one of the same people who did 4.7. It's not surprising that most people couldn't do it, as it follows on from 4.7 and requires plan no 11 again. Of the wrong answers, all except one were classed as "no attempts". The one program which was analysed was categorised as incorrect code: in fact the subject solved it by "cheating" and answering a different problem.

Exercise 4.9

This exercise required subjects to write a program which sounds the buzzer whenever a key is pressed. The following flow chart is given:

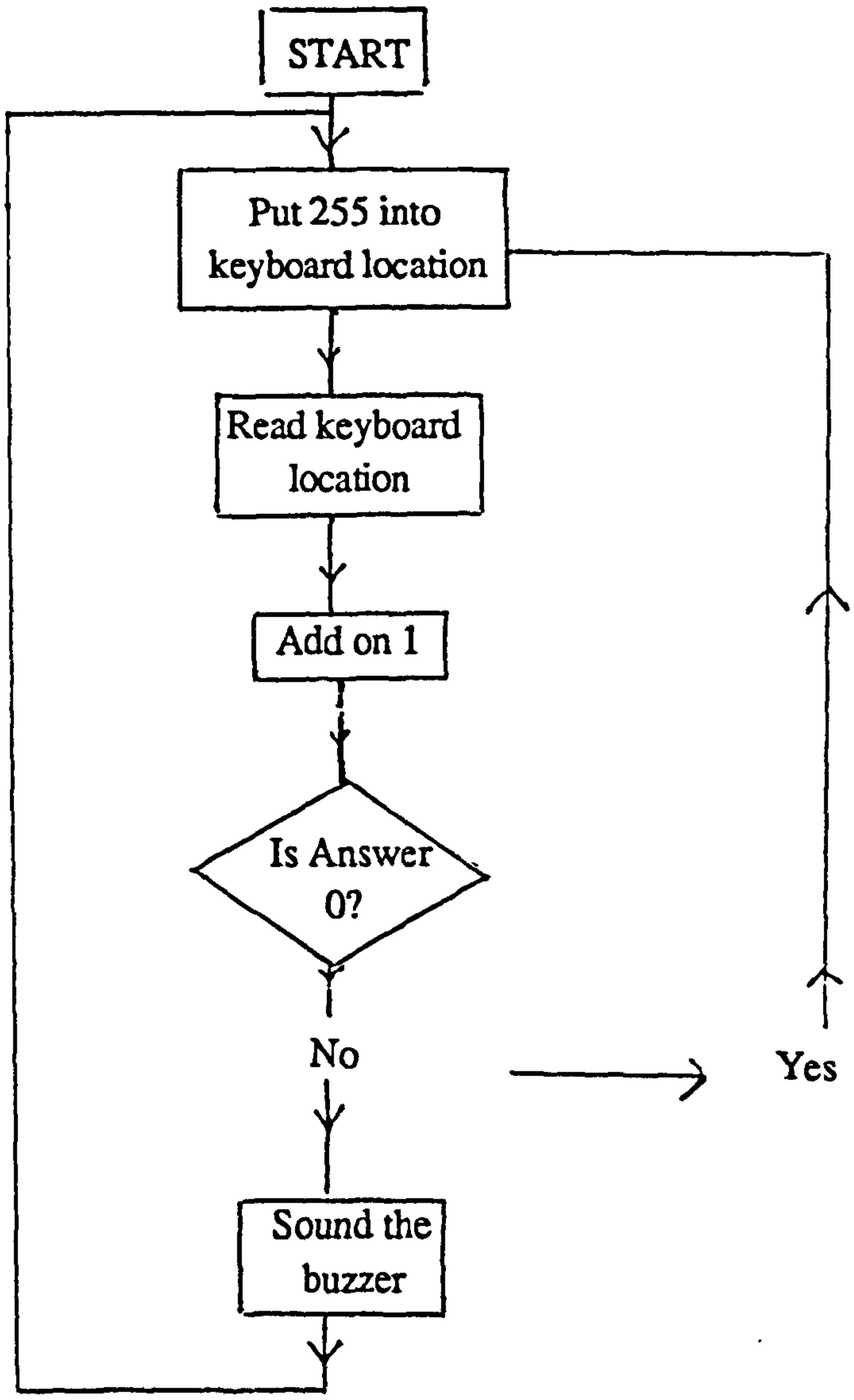


Fig 9.1 Flow chart which accompanies exercise 4.9

The flow chart is explained, and in the exercise the student is asked to write the program - with some help:

"Sounding the buzzer involves a) turning it on b) a short delay c) turning it off. My solution involves 14 lines of program so don't imagine it is very short. Use pencil and paper to write down your program and add comments to it otherwise you will quickly forget which part of the program did which job. When you are satisfied with your answer then read on."

Three people were successful here, but others commented that they did feel they

knew what they should be doing even if they didn't get it right. There are three common problems:

1. How can the count be achieved given that the accumulator will be used in the process and the keyboard routine also uses the accumulator and therefore will overwrite it? The count plan (7) was only mentioned in passing, and so I think one problem is that people are not familiar with it and have not had any practice using it. It does indeed use another location.

2. The flow chart suggests counting as soon as the lamps are turned on yet it's better to do the test before the count.

3. The given solution does not involve a rogue value for the keyboard routine, although the text emphasised that the keyboard plan involves this rogue value[3], this is the first instance of its use after its introduction, yet it doesn't use it! This is very confusing for building up a generalised 'plan'.

Both the two incorrect solutions which were amenable to analysis were very incomplete, and consisted of the keyboard routine only.

Exercise 4.11

The next programming exercise of interest is 4.11, as 4.10 is an extension of 4.9. This exercise requires a program which will turn on a lamp, start counting and display the lamp as soon as a key is pressed. It can be seen as consisting of plans 7,

[3]The text explains that a "rogue" value needs to be put into the keyboard location so that it is clear exactly when a key has been pressed. Otherwise if a zero is in the appropriate location it indicates one of three possibilities: 1) that key [0] was the last key pressed, 2) that [0] is still being pressed or 3) that no key has been pressed since the program started.

10, 2, 12, 11, 7, 1B and 2 in that order. Again the given answer departs from the keyboard plan and it also gives a different version of the delay plan. These problems were reflected in the answers given. Nobody gave a correct solution, and the three incorrect solutions contained the following errors:

1. Missing code (The "count" routine was not set to zero at the beginning)

2. Flow-of-control (Missing test from a loop)

3. Flow-of-control (Did not know where to store the "count" - and it got overwritten).

These types of problems illustrate the difficulty in deciding whether it is a flow of control problem or a plan problem. If the plans were better understood, - or understood at all, their use would include appropriate flow of control. However, the problems of flow of control are not treated in the text at all, so regardless of the plans, subjects had real problems in this area.

The programming exercises in chapter 5 are 5.2, 5.3, 5.4, 5.7, 5.9, 5.10 and 5.11.

Exercise 5.8

The first exercise where the errors have been categorised as plan related is exercise 5.8:

"Write a program so that the motor acts as a counter, moving one position each time a key is pressed."

This exercise and exercise 5.9 can be looked at together, since 5.9 is an extension:

"Rewrite the program to display the total number of key presses."

A possible answer to exercise 5.8 is:

```
00  LDA 255;          Put in test value
02   STA 105;         Check keyboard
04  LDA 105;
06  AD1 001;
08  JZ 004;
10  LDI 000;         Key pressed so move motor
12   STA 103;
14   LDI 001;
16  STA 003;
18   JSR 215;         Pause
20   JMP 000;         Repeat
```

Of the two incorrect answers, one had flow of control errors (a jump back to the wrong line and unreachable code) and the second was very hard to disentangle but started with checking the keyboard without a rogue value:

```
00 LDA 105
02 STA 103
04 JSR 200
```

and then sending the keyboard value to the motor followed by the denary display. However, neither the code for inspecting the keyboard (line 00) or operating the motor (02) is close to the plan given. On discussion of the exercise she explains her first line:

"it says write a program so that the motor acts as a sort of counter. Every time a key is pressed the motor should move one position which seemed to me a very straightforward thing, all it wanted you to do was to load something into it in order to... that every time you pressed a key it would move, to my mind, it didn't ask for those sorts of details, only testing as such."

She has missed the point about the rogue value, or forgotten about it, and indeed, her program didn't work properly.

Exercise 5.9 requires the total number of key presses to be displayed which involves using the count schema. There were two incorrect answers here (and 4 no attempts): one of these was discussed earlier (this is where the count routine is set to zero at the beginning and therefore can only ever get as far as one). The other one is very incomplete, and the author comments:

"I can't work out how to get the total number of key presses shown on the display. Where is there a symbol for how often I had pressed the key rather than the last key I pressed?"

5.10 had no schema related problems, and no attempts were made to answer question 5.11, which was:

"Write the shortest routine that you can that would make the motor move one step. This could then be used as a subroutine in future programs."

The problem here was that no-one could think of a shorter way than the one already given:

```
1 LDI 000
2 STA 103
3 LDI 001
4 STA 103
5 JSR 215  (pause)
6 JMP 000  (repeat)
```

(As the text states that moving the motor requires the sequence 0,1,0,1,to be sent to address 103, and a pause is needed between each sequence of 0 and 1 it is not clear what a shorter sequence would look like).

Using the wrong instruction

One of the errors made in the in-text exercises and problems was confusing two instructions such as LDA and LDI, or STA and JSR. In the former case, it could either be a conceptual misunderstanding arising from the problems discussed in the last category, or simply a slip. In the latter case, it may be that as STA refers to storing something at a device (such as the lamps) it implies a 'going to' the lamps, which is also true of 'going to' the subroutine. It may be therefore that these are conceptually confused. There is no evidence, however, that people are confused about the meaning of the two.

All the problems discussed so far, are in the first group. Three types of problems have been discussed: flow of control, plan related and wrong instruction. They all occur in programming tasks and are all programming problems. It is not possible to decide whether the confusion between two instructions is more likely to be conceptual or a slip. For the two other categories, however, it is likely that there is relationship between the errors and DESMOND's conceptual model and instructional design.

In chapter 4, DESMOND's conceptual model was discussed, and it was seen that there is emphasis on both a procedural and a representational view but little on a functional view. This is consistent with the problems subjects had in using plans, as plans require an emphasis on the functional view. The emphasis on the procedural view should facilitate following flow of control, but it will not necessarily help learners to get their flow of control right when writing programs, and there is no direct treatment of flow of control in the instruction, so again it is not surprising that there are problems here.

The next two types of problems are those concerned with problems mentioned in reading the manual operating DESMOND and trying to understand it.

Confusion between two terms or concepts, e.g. address/memory location

A number of people experienced problems with locations, addresses and contents in the first two chapters: problems which were often not reflected by their answers to the exercises: i.e. they completed the exercises successfully but didn't feel they had completely grasped the distinction between the above terms. The first concepts looked at are address and content. The two need to be seen together. The first chapter tackles addresses - and by doing this, their contents. Below are some of the problems mentioned:

1 Not being able to understand the ideas until they were applied in the activity

2 Indirect addressing:

"Suddenly when we called it the memory map devices those numbers which were what went into those addresses have become the address. I find that a difficult thing to understand." (S17)

The text does not specifically discuss indirect addressing but says:

"..some of the devices - like the switches and the sensors are memory mapped. This means that they replace certain memory locations and as far as the programmer is concerned can be addressed in the same way as User Memory."

3 Understanding analogies

Another person (S30) had problems understanding the analogy which introduced the ideas and seems to have interpreted the contents to be the addresses:

"I still do not understand how the hell a postman would deliver to these houses where they're numbered 10, 7, 255 and 0."

4 Inconsistent use of terms

One person, (S33), complained that the term address and memory location was used inconsistently and that address and location was used ambiguously yet in fact the terms are used totally consistently. The text he is referring to (page 85) says:

"..when you started ..in Single Step mode, DESMOND went to addresses 0 and 1 to find the first instruction. Once it had executed this, it went to addresses 2 and 3 for the next instruction. DESMOND will always move down to the next pair of memory locations until it is directed to do otherwise. For instance JMP 000 made D take the next instruction from addresses 0 and 1, and so start to repeat the program."

S16 also talked of lack of consistency and said that in the first chapter, up to (location) 99 were referred to as addresses/locations and thereafter as instructions/contents whereas in chapter 2, both are addresses and locations. Indirect addressing is also mentioned again in this chapter by S17. To understand the issues and problems connected with addresses, locations and contents, it is necessary to see how these concepts are introduced and taught.

In chapters 1 and 2 a strong distinction was made between a location (which can be identified by its address) and its contents (the value), e.g:

p16 In computer language we shall use expressions such as:-

The memory location with address 0 contains 10 .....

p16 The content of a ..memory location is often called data.

and later:

p19 It is possible to move to adjacent memory locations by pressing the arrow keys [<-] and [->]....The display shows the address 32 and the contents of that memory location. It is still 127....

The display is thus a window into User Memory and can be thought of like this:-

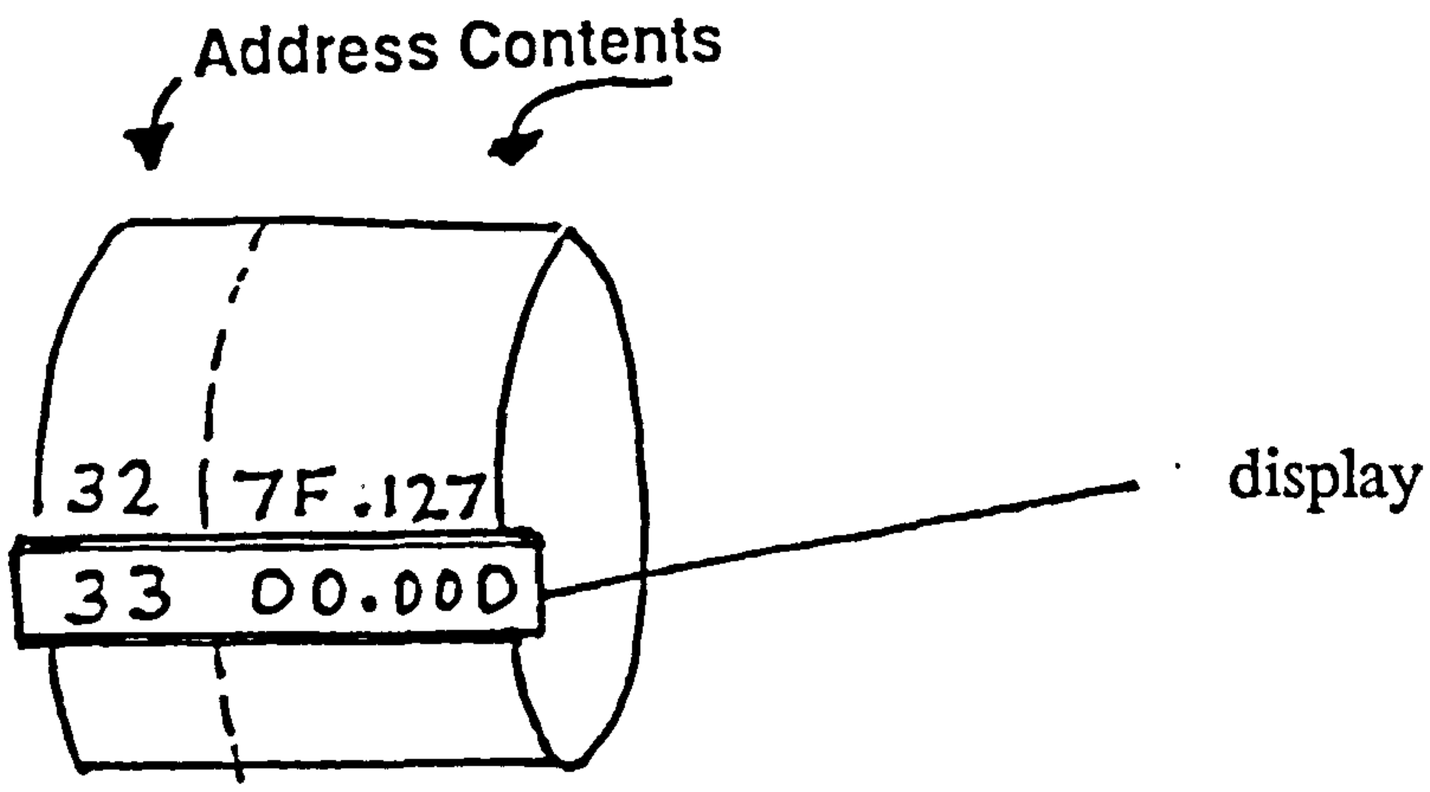


Figure 9.2: How the display is shown as a "window into User Memory"

In this chapter then, the location is identified by the address, and whatever is contained in that location is the contents. The memory locations themselves are always referred to as locations. The first place where this might cause confusion is the introduction of memory mapped devices. Here the dichotomy of location and contents breaks down. (Note, that in any case, for some people the terms address and location seem to be interchangeable anyway.) The memory mapped devices, although they have been used in a program in chapter 1, are first discussed on page 49:

"..they replace certain memory locations, and as far as the programmer is concerned can be

addressed in the same way as User Memory."

How does this relate to the distinction that has been set up? Is it consistent? If we take one of these devices, say the 8 switches, which are at address 104, this address needs to be referred to as the contents of another address. (This is a problem of indirect addressing, rather than one caused by the memory mapped devices, but it is in using the memory mapped devices that the problem surfaces). This is not really made explicit in the text, which says:

" The memory mapped devices replace memory and use addresses 100 to 199.....104,106,107 and 108, the numbers that changed the program between the switches and the sensors, are the addresses for these devices."(p51)

So, in chapter 1, the clear location/content division has broken down somewhat, in that the real picture is closer to this:

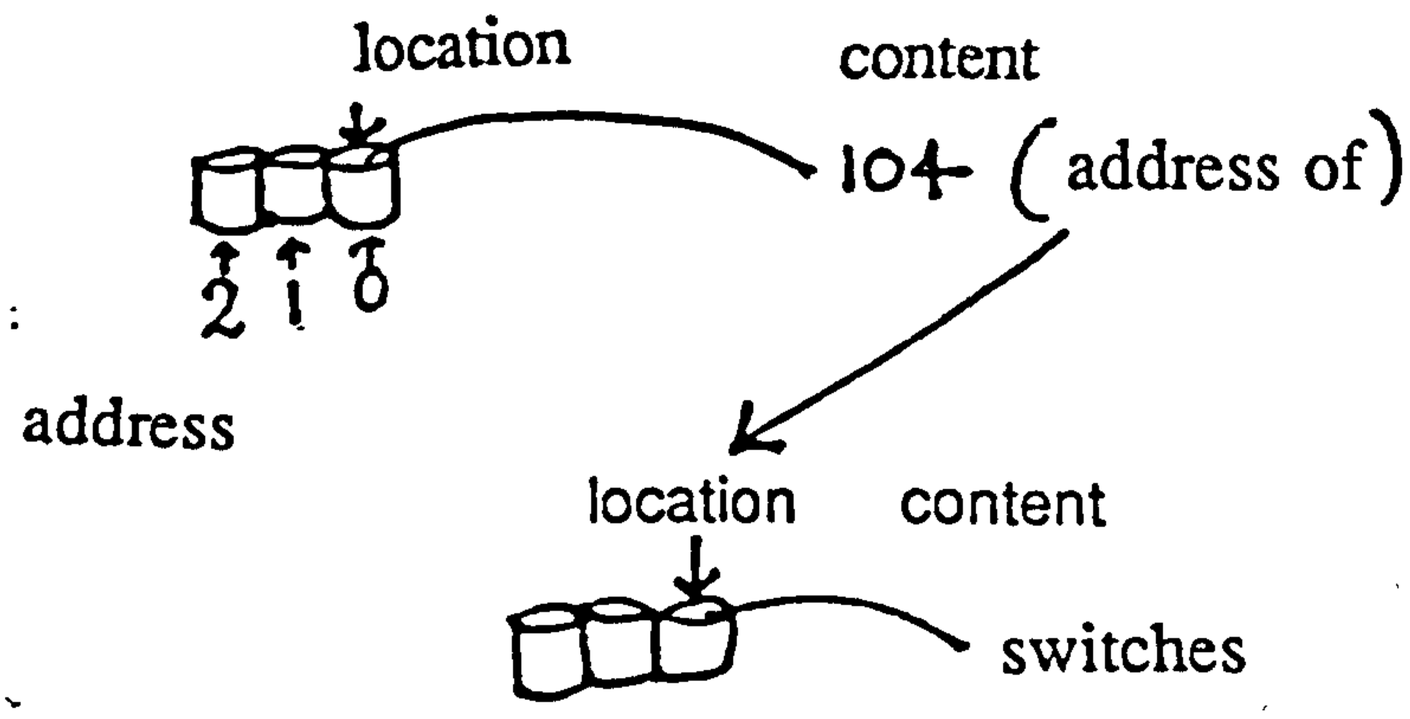


Figure 9.3: Location and content

In chapter 2 instructions are introduced. These also have two parts, somewhat

analogous to the memory location and content distinction. Here it is an operation and an address. For example:

"Load into the accumulator a copy of the contents at location 104 - :015 104."

When we get further into the explanation, we need once more to refer to addresses and now we need to remember that the content of an address is an instruction, with its two parts, the operation and the address. Let's expand the earlier diagram:

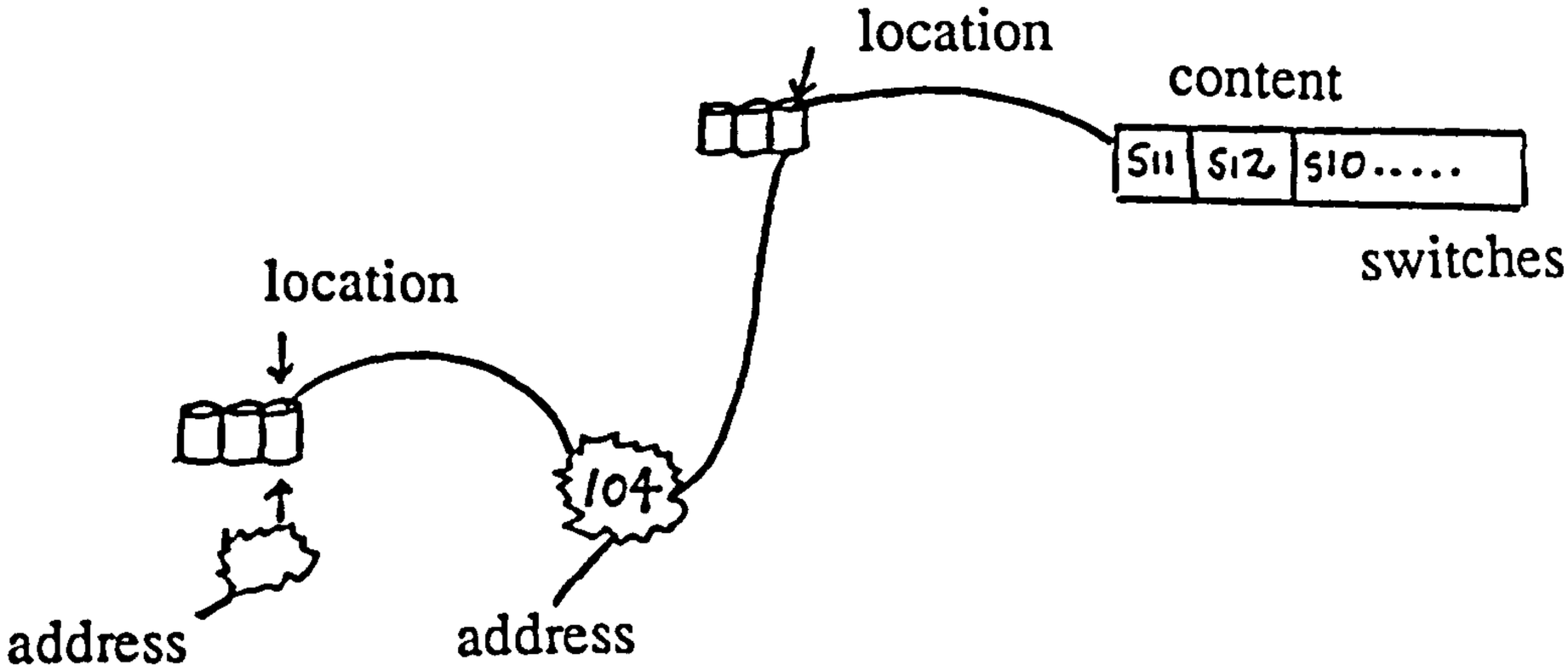


Fig 9.4: An expanded view of location and content

The diagram above could be further expanded to illustrate the link between the address/content distinction that we first met in relation to locations, and the operation/address distinction that relates to instructions (i.e. how an instruction is executed). One problem is that the two are never explicitly linked. (This could be done in a diagram.) The first exercises that follow (2.1, 2.2) aim to consolidate this distinction - presumably to give a firm grounding for what follows: however, they are perceived as being somewhat trivial, i.e. they are found to be very easy, requiring little thought, and subjects did not see the point of them. In spite of successfully carrying out such exercises people were confused.

A related issue is distinguishing between ADD and ADI. In one instance a subject was trying to sort out the distinction between ADD and ADI, and had keyed in the

first line of her program, which was:

00 ADD 001

02 .....

which enters ADD in address 00 and 1 in address 1. She had already written a previous version of the program with this first line:

00 AD1 001

This is unfortunate, because in this case the effect of the two different instructions is exactly the same![4] In both the examples given, the two parts of the instruction, the code and the operation will be in addresses 00 and 01. Although the code is different the result is exactly the same, as in the first example, the instruction ADD is an instruction to look for the code at the location with the given address which follows (ADD to accumulator from memory location 001). The given address is 01, and this also, of course is entered into address 1 (as it is a two byte instruction), so that address 1 contains 1, so 1 is added to the accumulator. In the second example 1 is added directly. The result of both lines of course, therefore is exactly the same.

Although this is an unfortunate and unusual example, it illustrates the kinds of problems that can occur. The idea of writing two versions of the program in order to explore the distinction between the two instructions was a very sound one. However, in this case, it had the same result, and so was not able to debug any

[4]ADD N is read as "ADD to the value already in the accumulator, the number stored at the memory location with address N." ADI N means "ADD to the value already in the accumulator the number N".

mistaken notions, and there was not other help available.

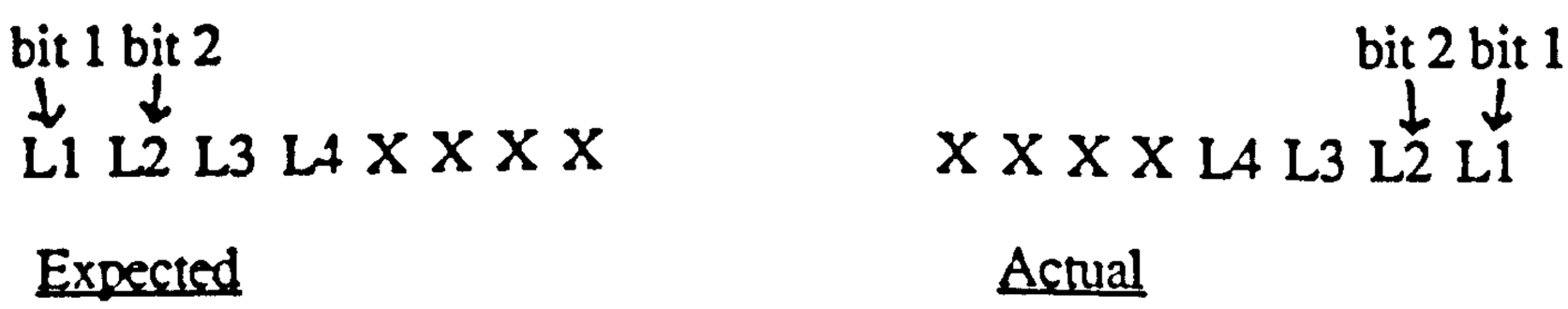
Confusion between two operations and understanding how a particular operation works

The example looked at to illustrate this will be the DESMOND routines. The first example of this is in chapters 1 and 2 which introduce some DESMOND routines. The very first program in chapter 1 displays the state of the 8 switches simply by displaying the contents of the location to which they're memory mapped, - location 104. There were some serious worries at this stage: namely that the switches don't affect anything (as indeed they don't).

"..it seemed odd that there were these switches where you could find out whether they were on or off but it didn't seem to make any difference to my performance on the computer whether they were on or off. And so I couldn't understand why, what are the switches doing. I could use the computer regardless of whether the switches were off or on so what function did they have? "

This seems to stem from an expectation that a switch being on or off should control or affect something - and later, of course, this does happen. The program is explained in chapter 2 (it is used as a demonstration in chapter 1). It is then used to control the lamps, i.e. information about the state of the switches is sent, via the accumulator, to the lamps, and as there are 4 lamps only the last four bits are used, and therefore only switches S11 - S14 control the lamps.

One of the problems encountered is remembering how the devices are memory mapped: both the lamps and switches are counter-intuitive in that the farthest right hand bit is mapped to L1 and S1 whereas there is an expectation that the lamps might be mapped from left to right, e.g.:



Subjects often got confused by this, and commented that they didn't really understand what was happening: they also forgot that only the last 4 bits are mapped, therefore only 4 switches can be connected, and later that for the buzzer only the very last bit matters:

S7(Ch 2) " When I programmed the lamps I didn't feel I really understood it."

(S8, Ch 2) "In 2.18 why is L1 operated and not the others "

Again, the reason is the mapping.

At the end of the chapter the state of switch routine is introduced as part of the burglar alarm. This works by loading the accumulator with a number which corresponds to whichever switch one wants to inspect before going to the routine. The accumulator then holds a 1 if the switch is on or 0 if it is off. Moreover the fact that the accumulator holds a number if the switch is on can be used to control a device - so that effectively a chosen switch controls a device. However, this was not only confused with the use of the location 104 earlier, but also caused problems in that people forgot that a 1 was put into the accumulator if the switch was on. For many people, therefore, how the state of switch routine worked was not at all clear, - and I think this may be because the contents of the accumulator are changed without that change of state being emphasised. Later in looking at the plans, another reason occurred: this problem uses a plan which has not been explained and is an amalgamation of two earlier ones and therefore may be confused with them.

In chapter 3, problems still occurred in working out the mapping of the switches and the buzzer, e.g. in Activity 3B, "Altering instructions in Assembly mode", the text reads: "In particular decide which switch controls the buzzer. The answer follows so try to think about your solution before reading on." S2 said:

"I knew the buzzer would work but not which switch controlled it. The following diagram was helpful but subsequently I couldn't work out which switches controlled what." S6 said: "I worked out that the switches controlled the buzzer, but couldn't find out which switch to control it. .....I really couldn't work it out, looking at the diagram it was simple, I felt stupid"

Another student didn't comment on this in the text, but didn't know which switch controlled the buzzer when doing the additional questions.

Syntactic errors

These are low level errors rather than conceptual, although of course they may lead to conceptual errors. Examples of this category are:

- forgetting the syntax or design of DESMOND (e.g. the meaning of brackets, which in this version of the text denote keypresses. This therefore leads to short lived mistakes in operating DESMOND

- errors in operating DESMOND (for example wrong keypresses)

- difficulty in locating parts of DESMOND (this really only applies to the early chapters)

There is another more interesting example however, which is to do with DESMOND's conceptual model - although it only applies to the earlier chapters (probably 1 - 3). Some people had problems remembering how to enter, check, and alter instructions in assembler, and commented that they found it difficult to remember when to press ACC (the accept key) and when to press the arrow key [>], and what part of the display will move when the arrow key is pressed. How Desmond operates in this respect is discussed earlier in chapter 4. Because of its limitations, it does impose quite a memory load on people. To enter the first line of an assembly program, for example 00 LDI 91, requires the following steps:

Action	Display	Comment
Press C	BYTE OOO	Choose computing mode (from the two courses)
Press A	ADDR 000	Choose Assembly mode (Points to first address)
Press ACC	(00).NOP.	Address 00 flashing in left segment of display
Press ACC	00.(NOP).	Middle segment flashes
Press >	00.(LDA.)000	Moves to next code (LDA) and still flashing
Press >	00.(LDI).000	Moves to next code (LDI)
Press ACC	00.LDI.(000)	LDI accepted. Right segment flashes
Press 9	00.LDI.(009)	Number enters from right
Press 1	00.LDI.(091)	
Press ACC	(02).NOP	91 accepted. Moves on to next address

This is 10 keystrokes in all. Clearly the aim is to make the current state as obvious as possible. The inconsistency of the use of the arrow keys was discussed in chapter 4. Consider entering the number 91. At other points where the display has been flashing [<-] or [- >] has been pressed to move through the options. In this case this would be uneconomical, (as even with the ability to move in both directions one would need up to 50 arrow presses!) and so 91 is keyed in directly. But it does mean that different operations are needed when doing apparently analogous tasks from the same or similar states, and this raises the number of things one needs to remember. Some people forget where they are, or how to get

back to a different state: "After pressing R and ACC how do you get back to the program for alteration?".

Four problems have been discussed in this section: confusion between two instructions; confusion between two terms or concepts; confusion between two operations and understanding how a particular operation works and syntactic problems. As with the first group of problems, there is a relationship between some of these and the conceptual model. The distinction between the locations and contents was emphasised, which is helpful in giving a clear picture of the machine, but this distinction does not address the concept of indirect addressing. It is in the use of memory mapped devices, which involve indirect addressing that subjects become confused. In chapter 4, the use of the arrow keys was discussed, and it was pointed out that it was violating the consistency principle. It seems that this does lead to some confusion.

9.6 GROUP 2: INSTRUCTIONAL

Jump in conceptual level

In several places the difficulty level of the text suddenly increased. Chapter 2 was the first place this happened. Most people found chapter 1 straightforward and chapter 2 much harder:

(S29, Ch 2) "When I programmed the lamps I didn't feel I really understood it. Activity 2c was complicated - a jump from chapter 1, ...It seems understandable, then you do something and you realize it hasn't sunk in. ... Felt I needed to know how to cut corners and get on with it because I seemed to have to keep going back and looking things up when really I wanted to know quickly "

5 other subjects made comments which indicated how hard they found chapter 2 compared to chapter 1:

"I must have put in excess of 10 hours on chapter 2. I was at one stage going to throw the lot through the window....(Page 67, (Activity 2A: Analysing a simple program) total confusion.

There is so much information coming at you all the time, it is hard to sort out where you have problems after a while or how you've solved them ."

In particular, people did not feel ready for problem 2.8.

Subjects reported a number of problems are mentioned in chapter 2 and this may make it seem particularly hard. They include data/locations, memory mapping of the switches and the switch routine (see previous category), the difference between JMP and JSR (which, although it is used for DESMOND routines, has not yet been explained), and between "go" and "jump" and LDA and LDI, single stepping, the switch routine, and binary number and binary pattern.

The next main point of difficulty is in chapter 4, where subjects again spoke of a sudden shift in difficulty level. For example, S28 said that "the course has suddenly jumped from being ridiculously easy to being very hard." S26 complained that the exercises differed too much: i.e. they needed more practice on similar problems before trying to apply the ideas:

" the exercises differ too much, so you never get the chance of doing essentially the same thing again in a slightly different way: it's always the challenge of something slightly different" .

As was discussed under plans, there was sometimes inconsistency between the plan which was introduced and the first example of it in use. The delay was illustrated as a backward jump, and the first instance of it in the answer to a problem was as a forward jump:

"What you have to do the next time differs slightly. One time it jumped backwards, .. the next time they expect you to see what isn't in fact the same, it's a jump forward. It's not that simple to do it differently without any warning. .... Once I saw it I thought well it's going to be the same, but I didn't think it was sufficiently alike to be able to cope with it." (S17)

Another issue was the jump between the text and the exercises, i.e. that the exercise required the use of techniques or concepts just learnt, but was significantly different from the examples given:

S24 (Ch 4) " This is where I seemed to have most (problems)..where you've done a similar kind of thing in the exercise before and then they ask you to do something related to that but obviously slightly different. I can see it with the answers but if I put that away and go back and try to tackle it I still can't relate it."

....

(E) So there's a bit of a jump from the exercises to the problems?"

(S24) I can't seem to bridge that little gap, to get from the exercise to the problem."

Sometimes the "jump" was the introduction of a new plan, as in plan 11, discussed earlier (testing for numbers other than zero). This came up in exercise 4.7:

"4.7 was particularly difficult. I read through thinking I must have missed something....."

Where a flow chart was given, or the ordering of the program, as in exercise 4.9 it was much easier:

(S24) "In 4.9 I had no problems,...the flow chart really helped and the steps like "sounding the buzzer involves turning it on" so you know you've got to start there, then there's the delay and then you've got to turn the thing on."

Where to start

One of the reasons why the flow charts help is because they get over the flow of control problems - to some extent. S24, quoted above, had recurring flow of control errors, and so it is not surprising that she found the structure of flow charts helpful. Often this expresses itself as not knowing where to start. S24 again said:

"Similarly in chapter 5 I couldn't do one and then the next one they gave a few hints. It's just knowing where to start."

Another problem in knowing where to start, apart from flow of control problems is that subjects do not feel they have had sufficient practice or instruction in working out problems from scratch. We have already discussed the problem of applying what has been given to a different problem and this is extended to creating a program. This is most evident in chapter 4 where several people said they could alter existing programs, they could understand the examples, they could

understand the answers to the exercises but they couldn't create one from scratch. Other comments included "fitting it altogether'.

One person said (S34):

"The instructions failed, from about page 165 on, to show me how to write a program"

Although this person was particularly negative - probably due to her lack of confidence - her comment is valid. The process of designing and coding a program has not been explicitly taught. One problem is the lack of examples. In order to successfully complete exercises or problems it is necessary to have either learnt (i.e. internalised) plans that can be used or know that they're applicable, find them and apply them. It's clear from previous discussion that subjects were often not in this position. Also, the question of flow of control had not been explicitly discussed and taught, and so not surprisingly this led to the problem of where to start: i.e. which bit should be dealt with first?

Expectations about the task

One problem in some places was not having a good idea of what the solution might be. For example in exercise 4.5 "Write a program that will multiply a value stored at.....Do not add the number to itself 16 times, try to think of a neater method."

This suggested to people that the answer should be quite short (which it is not), as well as mis-cueing them. Other exercises involved long answers like 4.8:

"I didn't get a complete working answer to this one... I was trying to find a much smaller program, .. I didn't expect it to be so long: I might have gone on much longer if I'd realised. " (S31)

Other people also said that they gave up when they might have been successful because they didn't know what the scale of the enterprise was:

"7.2 was O.K eventually. It said it was longer, which helped." (S20)

Expectations about the level of understanding

This category is concerned with the notion that the exercises can be completed without understanding. There are quite a few comments from people that they felt they were following instructions without understanding - especially in the earlier chapters. For example in chapter 1:

"Will it be made clear? Well that's just me with my question. I can do things mechanically, if you see what I mean, I can make, .. the instruction manual is written well enough to actually get into it and do what I am supposed to do and see what I am supposed to see but it is just me and these numbers....never mind."

and in chapter 2:

"It's a recipe. I can follow it unless jargon comes up so it causes a hitch but I haven't got a feeling of what the machine's doing - still absolutely none."

"As I understand it it's all to do with bits and bytes.....and I don't think ultimately that's what we're meant to understand" (S17)

Another comment was that subjects thought they had understood the text, but subsequently discovered they hadn't, or at least they couldn't apply their knowledge:

"it was thinking I knew and finding out when it was repeated that I didn't know. When I programmed the lamps I thought I understood it, - and started re-reading (p70 - 78). This is where I realised I hadn't understood it. I went back to the beginning and stated again." (S32))

"You read it and it seems understandable and then you do something and you realise it hasn't really sunk in."

In the earlier chapters it's understandable that people feel the newness and strangeness of the domain, and of course at this point they haven't yet done any programming. However, the view that the understanding that they're acquiring is not at a very deep level is repeated, and is to a large extent borne out by the performance on the exercises. The best way of summarising this is to say that there's a gap between people's comprehension and their production. In the same

way that proficiency in understanding a new natural language is far ahead of producing that language, comprehending the DESMOND text, following the examples and understanding the answers to the questions was far easier than writing programs. In fact, by the later chapters some people had given up trying to do the exercises but tried to understand the answers. This gap frustrated people:

"Where they say they want you to do this and this is the sort of program,... I can do those, no problem, .. but actually creating a program, I can't seem to be able to tackle from the beginning."

9.7 GROUP 3: AFFECTIVE

The final group of problems is in the affective area.

Expectations about learning DESMOND

In chapter 3 it was argued that computers may be qualitatively different from other devices that people know about. This is partly to do with their range of behaviour - and it was argued that it affects the mental models that novices develop and hence their learning. In this study the subjects did not have a clear idea of what learning about computers means, and they didn't know at what level they will need to understand and operate, yet they sometimes expected to understand at a more detailed level than is necessary.

Many of the subjects who answered the advertisement had their own particular agendas for why they wanted to participate, - yet as they got under way it seemed to me that their objectives weren't achievable. Their reasons for taking part in the project included comments like:

"To begin my understanding of the concepts and capabilities of computers".

"I feel that I am sadly lacking in knowledge, and hope that this project will set me on a course of

learning more about computers."

"To have some hands-on experience with some structured material to guide me, i.e. to give me some confidence about computers."

Most of these aims are rather general, although there were some people who had specific interests in programming.

When they came to learn DESMOND people often felt frustrated by not having a particular application in mind. What use was learning to put things in and out of memory locations? Yet as they did not have a particular purpose in mind (such as learning FORTRAN to process your PhD results) one could argue that they would find any general curriculum unsatisfactory as they wanted the course to be applied and have a clear purpose, but they did not have a task to be done! Strangely, it was often the less knowledgeable subjects who expected to understand at the most detailed level. The quotation below illustrates one subject coming to terms with what is needed. S15 realised that he didn't need to understand binary, hex and decimal as long as he understood the principle:

S15 (Ch 1)"...I didn't need to understand them as long as I realised that there were two ways of being, doing figures if you like, or counting fingers and one is one that I'm used to and one I've never seen before in my life."

S17, probably one of the least confident, was worried by terminology she hadn't met before, and being so anxious that she didn't know what prominence to give to something she's not sure of:

"the picture was very gungy: you really couldn't see where the off/on switch was"

i.e. the reprographed picture was not very clear: although in fact it's not very hard to work out as it's clearly marked on the DESMOND. She also interpreted the keypresses incorrectly,- pressing A - C - C instead of ACC, and was worried about how much she needed to remember:

"I'm not sure whether I'm supposed to be remembering: I can remember how to get a program started and to change it and that sort of thing, but am I going to have to bear in mind what...the input of the address is in order to change.....?"

She was unsure of the level of knowledge and understanding required:

"It's like driving a car, it's handy to know what is happening rather than just doing"

but this is an interesting analogy precisely because most people don't know what is happening yet happily drive cars, operate central heating systems etc.... the point is that computers, to her, were so strange that she didn't know what she could comfortably not know! When encountering unfamiliar terms she didn't know at what level she needed to understand them, and didn't feel able to continue without that knowledge:

"And again, this is the chapter with a lot of jargon and so if you are unfamiliar and unsure of yourself after a while you come across things and you think 'oh for goodness sake' what's happened."

Her interview partner (S16) had quite a different reaction, although there were things which she, too, didn't understand clearly. In the extract below, S16 and S17 are discussing this problem:

(S16) "I suppose it depends what level you're taking it whether you're going to accept it and think well I sort of understand and I hope I will totally understand it at the end.
(S17) I'm sure that's wise..I get in a panic when I don't fully understand it but I get in a worse muddle if I go on without fully understanding it.
(S16) So much of my life I don't understand you see so I just have to carry on."

The DESMOND curriculum is being taught to people with no previous experience, and as a self study pack, so they have no teacher on-hand to advise them if they get stuck. The way distance learning designers usually cope with this problem is to structure the materials very carefully - and to guide students through at a very detailed level by telling them exactly which key to press when and why. This is a sensible strategy for novices as first of all this amount of hand-holding can overcome nervousness and secondly as long as people follow the instructions, if

they do have problems it is easier to try to diagnose their problems.

However, there are at least two problems with this approach. The first is that although the text can try to clearly state the steps that should be followed, - it can never be totally unambiguous. The author cannot therefore be assured that the instruction or intention will be understood by the learner in the way that the author intended. The second problem is that learning to program involves building up a repertoire of plans - and using bugs creatively to diagnose, and learn, and so it is possible that the style of pedagogy and the nature of the domain are in conflict. This category is concerned with the first kind of problem: what happens when beginners following a closely guided exercise meet something they are not expecting? There are two points here: the first is that because the text accounts for events in such detail any variation from expectation causes some concern; and the second is that individuals' tolerance of this varies considerably.

One issue is what to do if unexpected things occur, and this is connected with the curriculum. Unlike S17, S16 was able to keep going when unexpected things occurred, and usually they resolved themselves. S30 found that the test routine wasn't as expected, and didn't know whether it was his problem or not, and later a similar problem occurred again where his display was different to what the text had suggested because it was displaying the state of the 8 switches, and it therefore depended on whether an individual's switches were on or off:

" I thought why should I have a different pattern if I've filled in the things properly and it's all arithmetic and sums it can only be the same thing"

After explanation, he realised why this is the case. Examples further on in the curriculum are exercise 3.4 in chapter 3, where the draft copies (which everyone has) are only able to represent X by X, and this leads to problems in doing the

exercise. Here again, people coped quite differently with this. A final example is in chapter 4, where the text is discussing putting rogue values into the keyboard location, and the text is misread as suggesting that a two digit number can be entered into the location via the monitor mode, whereas in fact it needs to be entered as part of a program.

Learning to program

Finally there is some evidence that some subjects had pre-conceptions about programming. They were often surprised and frustrated at not succeeding on an exercise on their first try and did not expect learning to program to be a difficult, frustrating process. For example S28 commented that the course changed from being ridiculously easy to being very hard (in chapter 4), and in chapter 5 gave up on exercise 5.9 after 21 minutes, commenting that "it's still not working". It would seem that her expectation was to complete a problem successfully in 10 minutes or so. She gave up the course at a point when she was in fact doing quite well.

Lack of confidence

This speaks for itself, but can be illustrated by remarks in early chapters such as:

"given my lack of natural ability in anything like this - I've all the built up inhibitions that I have partly because I'm X's wife.....So I've got all of these anxieties"

and self-deprecating comments like:

"Being an idiot, I take everything I read literally, so if it says something different I begin to question whether I'm doing the right thing or whether the text is slightly wrong or the machine is wrong"

Other problems which don't fall into the above categories

Terminology was a problem: for example when it failed to map onto a subject's existing knowledge:

"Memory mapping for instance.. my concept of a map is the sort of thing which is a bird's eye view of something flat and I found great difficulty in understanding various bit of jargon, making in effect a change with the concept I'd had in my mind to what was apparently being asked of me was quite difficult." (S17 Ch 1)

or it may be described as jargon: "specified operation code?". The most commonly reported jargon or terms which were not understood were:

denary, binary and hex., specified operation code, data area, message buffer, ASCII, reserved data areas, flag, two byte instruction and read keyboard location. Problems of distinguishing between different terms were often mentioned, e.g. JMP and JSR, binary number and pattern, go and jump:

"sometimes the word go is used and sometimes there's jump to and sometimes there's jump back...and I don't know whether jump and go and jump back or jump to or go back are really all the same way of saying the same thing - I know jump the way it has been used here always seems to lead to repeat but does it always? ".

Misunderstanding (as opposed to claims of not understanding) sometimes resulted in exercises being carried out incorrectly: e.g. two people (S16,S17) started doing activity 2 (a) when they should still have been reading the text. In exercise 3.4 two people thought they had to enter the ASCII codes for the whole alphabet in locations 91 - 98! Another piece of text read:

"Suppose we put 12 into location 105. Next we have a loop to keep reading address 105 and whilst the number we read back is 12 we know that no key has been pressed. ....."

and many people tried to carry this out straight away, and then discovered, that as they expected, DESMOND bleeped if they tried (via monitor mode to put 12 into 105, because as a memory mapped device, it is not accessible in this form. The phrase **"whilst the number we read back"** also led to some confusion, and S9 commented:

"I thought I was supposed to read it".

In fact, she couldn't understand this part at all as she didn't know at what point the key should be or would be pressed, which obviously affects tracing through the working of the program.

Particular concepts or processes were sometimes not understood - for example binary addition in chapter 6 caused some problems for S26 - as she found the text to be in conflict with her own knowledge about how numbers behave:

"I had to re-read it very carefully 3 times before I started to understand what they were getting at.....Once you've accepted that nought and one whichever way round they come is nought, and 1 and 1 is automatically 1 it sorts of overrides everything you've ever been taught about one and one being two etc."

This discrepancy between existing knowledge and new information can lead to the formation of buggy mental models, as well as being the conflict which leads to learning! This was also discussed in chapter 8 of the thesis and is an example of the problem of misinterpretation where people are actively interpreting with little or no other source of evidence available.

9.8 CONCLUSIONS

This chapter has looked at the performance of two groups of subjects learning DESMOND. The subjects who had previously learnt LOGO (group 2) performed slightly better than the group who were learning DESMOND first (group 1). This difference, however, was not significant, and so for the rest of the discussion the two groups were combined, although possible reasons for the slight difference were discussed.

Problems which students encountered on both programming and non-programming tasks were categorised into the following groups: programming,

instructional and affective. In the programming group there are two main problems which are inter-related. The first is the identification of appropriate plans and their use: this problem is indicated both by the quantitative and the qualitative data, and the second is flow of control. In both these cases there is insufficient instruction on writing programs and making the plans needed to do so explicit and this is shown in the problem of not knowing where to start. Indeed the total number of programs available for analysis was far less than the total attempted - if by attempts we include mental struggles which did not result in putting anything on paper. It has been argued that there is a relationship between the problems students encoutered, and DESMOND's conceptual model with its emphasis on both a procedural and a representational description rather than a functional description.

Further problems in this group were manifest in non-programming tasks. Often these occurred at an earlier stage of the curriculum where there is less programming. They include: confusion between two terms or concepts, and between two operations, and not understanding how a particular operation works. Some of these are problem areas particular to DESMOND or assembly languages and include: memory mapped routines such as the state of switch (which may be a cognitive load problem and a lack of clear tagging of plans), the address/contents distinction and related distinctions such as LDI/LDA, ADI/ADD, and for some binary was a problem. This group also includes syntactical problems, and although these did not have a significant impact, it is possible that the design of DESMOND and its conceptual model imposes too big a cognitive load onto the student.

The second group is related to the instruction. It is suggested that 1) the style of pedagogy and the nature of the domain are in conflict. There is a mismatch between the distance learning guided hand-holding approach and the nature of the

domain, and it seems that this style of approach encourages people to expect it to be too easy at times. 2) The closed nature of the curriculum may also be related to people's lack of tolerance of ambiguity and any variance from their expectations, - especially in the least confident. 3) There are conceptual leaps - where the step between what had been explained and the next exercise, or an exercise and a problem was too large, and there was insufficient practice on particular points.

In the third group more affective problems are considered: for example there is also a variation in people's ability to tolerate incomplete understanding - again the least confident have most problems. Although they are of a different kind, these problems are not any the less serious: few teachers need persuading of the effect of believing one has a mental block or of a total lack of confidence - but by their nature these are less easy to document. They include the expectation that learning to program is straightforward, and lack of confidence, which leads to expectations that are not fulfilled.

There was much more information for DESMOND than for the other languages, and this resulted in the greatest range of problems of all the languages, with a number of problems in each of the three groups: programming; instructional and affective. This seems to be due to a combination of two main factors: firstly the methods used, which, (especially the filled in comment sheets), enabled a fuller analysis of the programming problems; and secondly the curriculum was good. In the case of Logo, although the same methodology was used, the curriculum was not so good.

There was less evidence in DESMOND than in SOLO or PT501 of instructional problems such as the text misleading the subject, but there was a different instructional problem of there being places in the text where the level of difficulty

suddenly increased. The instructional problems reported were consistent with an "improved PT501": for example some of the subjects felt that they had not been taught programming successfully, yet unlike PT501, they coped well with the mechanics. The methods used also allowed affective problems to be reported. In PT501 there was clearly a number of affective problems - as indicated by the attrition rate, but there was no opportunity to report this directly. This issue, of how the methods used effect the kinds of problems reported, will be taken up in the next and final chapter.

Chapter 10

Conclusions

Contents **Page**

10.1 ACHIEVEMENTS

This thesis has focussed on the problems which face novices when they are working in an environment which provides good conceptual models. The work is at the junction of two areas: instructional design in distance education and novice programming. Interest in distance learning is increasing rapidly, and many firms and institutions are investigating the use of open learning techniques. It is clear from studies such as those of Bott (1979) that the quality of stand-alone instructional material in computer related domains is very variable, yet this is an area where it is important that we get it right. It is generally agreed that learning to program is difficult. One way of helping novice programmers along this difficult path is to somehow help them to bridge the gap between what they already know, and the new concepts they are encountering. Several researchers have advocated the use of conceptual models as a device which does this. This thesis has examined four different languages in use which are taught via a conceptual model. Previous studies (e.g. Mayer 1975) have shown the efficacy of using such models, but none have investigated the use of such models outside the laboratory as part of the normal teaching curriculum, nor have they been particularly concerned with the curriculum used. This thesis has filled this gap by investigating the kinds of problems encountered by students using the four different languages, and has related their problems to the conceptual models presented and to the instructional design of the teaching material.

Chapter 3 provided a theoretical framework for the thesis, and suggested a further three ways of describing conceptual models for programming which distinguished three views of the conceptual model: state, procedure and function. This is not a categorisation system, but a way of describing conceptual models so as to highlight the different aspects which are important for the novice learner. It is a way of identifying the different kinds of knowledge which are necessary to understand the

conceptual model, and therefore a way of assessing whether a conceptual model highlights all three views or emphasises only one or two. In this respect it complements the criteria offered by du Boulay, O'Shea and Monk (1981). In chapter 4, which analyses the four languages and the way they are presented, it was seen that the criteria offered by du Boulay et al (which had not been empirically tested) were insufficient to evaluate conceptual models. In particular, du Boulay et al failed to predict the problems novices have in learning one of the languages they give as an example of good practice - the PT501 assembler. The version of LOGO used in this thesis was also problematic. In their analysis these environments are seen as exemplary, but a more detailed analysis of the instructional material using both the criteria offered by du Boulay et al, and the new classification shows that they are not. The various languages are presented in ways that highlight different aspects. For example, SOLO performed well on the criterion of simplicity, and quite well on consistency, but not very well on visibility; but in terms of the different views of the conceptual model it was seen that there is very little functional description in any of the languages. This lack of emphasis on the functional aspect has implications for students' acquisition of plan knowledge.

Whilst chapter 4 analysed and evaluated the languages and conceptual models, chapters 6 - 9 presented the languages from the learners' point of view: in particular regarding the presentation of the conceptual models. The main question being asked here was whether teaching programming in this way made life easier for the learners or whether they experienced the kinds of problems discussed in chapter 2. Studies of all four languages in use show that subjects did not develop the competence that might have been expected, although the problems they experienced varied from language to language, and encompassed both programming difficulties, and difficulties related to their learning approach. The different languages had different aims and curricula and engaged the students in different tasks, and to that extent each must be viewed on their own terms.

There are a number of problems which are general. Subjects' problems can be divided into two main groups: the domain itself, programming, and how the learners approach the learning task. *Flow of control* was a problem for students of three of the languages: SOLO, DESMOND and LOGO: there was insufficient data on PT501 to determine whether it was a problem here too. There is a great deal of evidence in the literature that flow of control is difficult for novices (e.g. Green, 1980b, Miller, 1975) and the evidence in this thesis supports such findings and suggests that good design practice (such as making all the conditions explicit in SOLO), and providing conceptual models, does not alleviate the difficulties completely.

Inadequate plan knowledge was a problem for SOLO, DESMOND and LOGO subjects. For example, SOLO, LOGO and DESMOND students attempted to use example programs, given in the texts, as models for developing their own programs. Yet many of them had insufficient competence to carry this out successfully, and such failures were related to lack of plan knowledge. In order to understand the difficulties subjects had in using DESMOND, the instructional material was analysed in terms of plans, and it was discovered that students need knowledge of 21 plans in order to complete the DESMOND exercises given to the subjects. As none of these plans were given explicitly, and their generic properties were not explained, the implication of this is that giving students the plans and making their function explicit would help them considerably. Also, this weakness is predicted by the analysis of the conceptual model made earlier, as plans require a functional view, which neither the DESMOND instructional material, nor the DESMOND machine gives. The finding that novices had these difficulties in recognising, acquiring and using plans is consistent with the literature on the use of plans and their importance in programming (e.g. Gilmore, 1988) and was also predicted by the lack of functional models available.

In LOGO, DESMOND and SOLO there were examples of subjects *endowing the computer with intelligence* (which is further support for Pea's findings (Pea, 1986)) which were discussed in chapter 8, and using their everyday knowledge to solve programming problems. Their everyday knowledge was also used in building erroneous models of the notional machine. These *inaccurate mental models* conflicted with the conceptual models offered. There was also evidence that the text can cue inappropriate expectations, and therefore facilitate the development of inaccurate mental models. The picture which emerges is that in the absence of other information (for example teachers or helpers being available), novices are reliant on the programming environment, - mainly the instructional text, to provide them with their information and knowledge. This information is provided via a conceptual model, which is not adopted as it stands, but used by the novices in developing their own mental models. In doing this, they use the information that they have available, - the screen, text, help and error messages, in order to try to confirm the hypotheses that they have set up. This view is consistent with Carroll and Mack's description of learners in a related domain (Carroll and Mack 1982). In other words the subjects in this study experienced some of the problems reported in studies where no conceptual model was provided. This is not as pessimistic as it may at first seem. While the problems certainly exist, they are not as severe as the difficulties reported in the other studies. Two reasons for these problems were suggested in chapter 6: firstly it is necessary for learners to take an active role in order to make their own sense of the information. In so doing they are inevitably going to develop inaccurate models at various points. Secondly, as they are using self instructional materials, they only have access to these materials (including screen displays, help messages etc) when problems occur.

Does learning a high level language facilitate learning a low level language and vice versa? The performance of the two groups who learnt DESMOND and then

Logo (group 1), and Logo and then DESMOND (group 2) was compared and the results were discussed in chapters 8 and 9. On DESMOND, group 2 (who had prior experience of Logo), appeared to perform slightly better, - but this difference was not significant. Turning to Logo there was less apparent difference between the groups and again this difference was not significant, although again the performance of group 2 was slightly better. So although the differences are not significant they run counter to the direction that would be expected if any facilitation or transfer occurred; so the case for transfer, on this evidence, must be rejected.

Another issue which emerged from the study was the relationship between the style and structure of the curriculum, the content, the attitude of the learners and the difficulties they experience. In line with most distance learning texts, all four texts guide the student quite closely by structuring the material very carefully, and in some cases (particularly PT501) requiring students to carry out experiments by following step by step instructions. There is some indication that this style of instruction may encourage the learner to adopt an over-dependent attitude towards the text. PT501's detailed hand-holding style requires that the learner has a great deal of confidence in the accuracy of the text. Yet when a problem occurs, the learner needs to decide what the nature of the problem is. She has to decide whether her interpretation of the text was correct, or whether her misinterpretation, or an error in the text, led to the problem. An example was given in chapter 7 of a subject "correcting" the text and compounding the problem. She was right to feel convinced that there was a mistake, but her diagnosis of the mistake was wrong, and there is little help for learners once they are in this situation. Subjects' tolerance of ambiguity in the text, or of incomplete understanding varied considerably. Where there were errors in the texts, or they had misinterpreted the text, they were often reluctant to accept that this might be the case, and some subjects' tolerance of ambiguity was very low, and they were

disturbed by any variance from their expectations. Even if such misinterpretations or errors in the text do not occur, there are further problems in working at this level. The PT501 experiment book, for example, adopts a step by step "recipe" type approach. One danger, therefore, is that as the experiments consist of very structured steps, they can be followed and yield the correct answer without a deep understanding of the process.

A further related issue is the nature of the domain itself, programming, and how that interacts with this style of teaching. In programming, a learner needs to operate at a relatively high level of abstraction, - noticing that different examples are using the same plan, interpreting and synthesizing. This view of programming is consistent with the emphasis on plan knowledge. It is not consistent with an over structured and guided style of instruction. An alternative goal or problem focussed approach, which is conducive to learners building and using plans, was suggested in chapter 7. In this approach, instructions would be introduced in the context of problem solving, and so their functions would be discussed and emphasised in considering questions such as: "When do I use the JUMP instruction?" and "What group of instructions will do the job required in problem X?" Such an approach would facilitate the development of semantic rather than syntactic knowledge and would focus more on the skills and methods required to solve particular programming problems.

The Logo tutorial manual takes a slightly different stance, which is part of the Logo learner centred philosophy. The learner is invited to explore and discover Logo in a somewhat less structured way. However, whilst this does not lead into the recipe problem that the PT501 manual has, it leads to a different set of problems and contradictions which are discussed in chapter 9. It can be argued that the PT501 curriculum is weak by being over prescriptive, whilst the Logo curriculum, in attempting not to be prescriptive, fails to meet the needs of the

intended audience, and is criticised both for misjudging the audience, and therefore presenting both inappropriate material, and using an inappropriate style, and also for containing too many inaccuracies. This is a particular problem in distance learning, given that students rely heavily on the text, and trust them to be right. Another factor that interacts with the distance learning situation is that learners actively interpret the text and form mental models. This is inevitable and necessary in order for them to learn and assimilate the material that they are encountering. The text needs to encourage readers to think and make connections:

"Although many (texts) are attractive, accurate, readable and understandable they are also one of the biggest deterrents to thinking in the classroom, because the writers assume that students learn best by studying a polished product. The key function of the writer is to explain, and a good explanation is interesting, orderly, accurate and complete. The vocabulary suits the level of the student and complex ideas are clarified by dissection, integration, example, and visual images. Thus, the textbook is weak in that it offers little opportunity for any mental activity except remembering. If there is an inference to be drawn, the author draws it, and if there is a significant relationship to be noted, the author points it out....."

[Sanders, R.S: Classroom Questions Jossey Bros, 1961]

The distance learning texts that were used are not like the "perfect text" discussed above, in that they contain plenty of activities and exercises, and in spite of the concern that learners become too dependent on them, it is clear that the subjects in the study were active in their learning, - interpreting the text and making connections. It seems, therefore, that the designer of distance learning or self instructional material for the programming novice is faced with a dilemma. The material needs to be structured and much hand holding provided such that the novice can confidently negotiate the material, but it should not lead the novice to

place too much reliance on the text. She needs to develop the abilities to solve the problems that will undoubtedly occur, and have faith in her ability to do so. However, encouraging (and indeed requiring) students to actively interpret the text means allowing them to make their own mistakes, form misinterpretations and incorrect models. That is what happened to many of the subjects in this study. The issue is not about preventing them from doing this, which is neither possible or desirable, but about how to support them when they have made mistakes.

To summarise, the thesis has achieved the following:

1 It has examined four different languages in use which are taught via a conceptual model.

2 It has investigated the use of such models outside the laboratory as part of the normal teaching curriculum.

3 It has related the problems experienced by subjects learning these languages to the conceptual models presented and to the instructional design of the teaching material.

4 It has provided a framework for analysing the presentation of conceptual models for programming which distinguished three aspects: representation of states, procedural and functional.

5 It has used this framework to identify the different kinds of knowledge which are necessary to understand the conceptual model.

6 It has shown that the criteria offered by du Boulay et al (which had not been empirically tested) are insufficient to evaluate conceptual models.

7 It has shown that subjects did not develop the competence that might have been expected, although the problems they experienced varied from language to language, and encompassed both programming difficulties, and difficulties related to their learning approach.

8 It has identified and discussed the following problems: flow of control, inadequate plan knowledge, endowing the computer with intelligence and the use their everyday knowledge to solve programming problems. This everyday knowledge was also used in building erroneous models of the notional machine. These inaccurate mental models conflicted with the conceptual models offered.

9 It has analysed the DESMOND instructional material in terms of plans, and it was discovered that students need knowledge of 21 plans in order to complete the DESMOND exercises given to the subjects.

10 It has shown that there is a complex relationship between the style and structure of the curriculum, the content, the attitude of the learners, and the difficulties that they experience. For example, it is suggested that the style of learning often used in distance learning may encourage the learner to adopt an over-dependent attitude towards the text and a low tolerance of ambiguity and errors. Further issues include the relationship between the nature of the domain and this style of teaching.

11 It supports earlier findings by Green (1980b), Miller (1975) and Soloway et al (1983) concerning the difficulties which novice programmers have, and additionally argues that the kinds of problems observed will depend on the conceptual model, the curriculum and the methods of investigation.

10.2 CRITICISMS

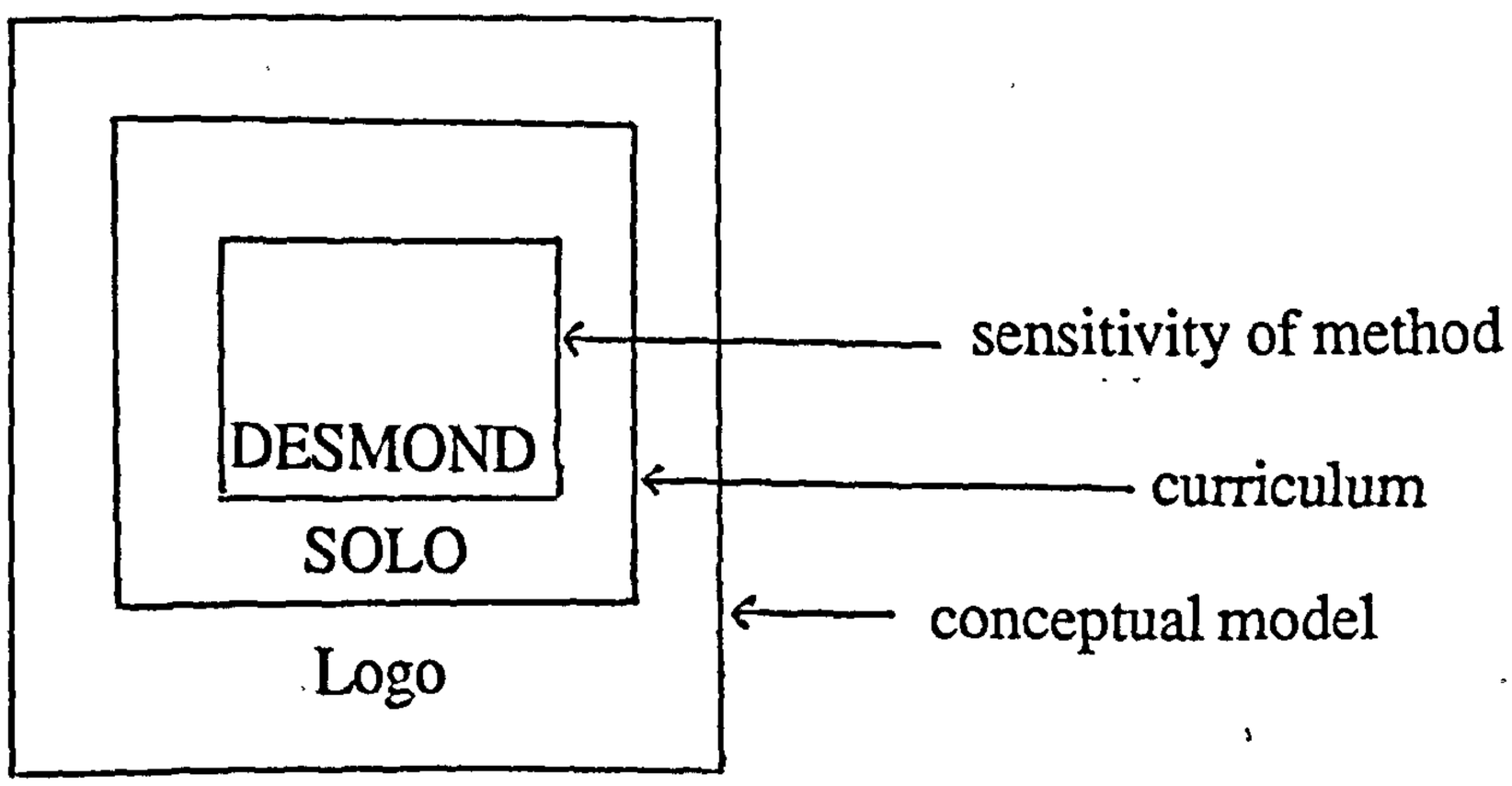
One weakness is the failure to answer the transfer question adequately: does learning a low level language help students in learning a second high level language and vice versa. There was insufficient data for PT501 and SOLO that was suitable for investigating this question. This was partly due to there being insufficient appropriate exercises, as discussed above, but it was also the case that few subjects developed competence in using PT501. As far as DESMOND and LOGO was concerned, there was no strong evidence that learning one affected learning the other; however, another problem was that no tools were developed in order to make a straightforward comparison. In any case, this would be a difficult exercise: all four languages are very different, they involve different tasks and there is no straightforward way of comparing what has been learnt across them. In order to do this, language-independent tasks would need to be developed and used.

Another issue is that the LOGO and DESMOND studies yielded more useful data than the SOLO and PT501 study. There are two main reasons for this. One is that the comment sheets which LOGO and DESMOND subjects filled in provided both error data and subjects' comments on the various sections of the instruction manual that they were studying. Whilst SOLO and PT501 subjects were asked to answer all the exercises in the books, they were not given comment booklets, but sheets of paper, and probably because these looked less "official" they often omitted to write down their answers or their workings out. Secondly, both the SOLO and PT501 manuals contained less exercises and questions that were useful in diagnosing the problems that subjects had, and because one aim was for students to be working with real materials and curricula, these were not supplemented. This was also true to some extent for LOGO.

The kinds of problems observed depend critically on three factors: the conceptual model, the curriculum and the sensitivity of the method of investigation. Thus DESMOND has the fullest analysis because the conceptual model is good, the curriculum is good and the methodology combines both quantitative and qualitative data (though not the on-line qualitative data which was present in the earlier studies and which may account for the lack of data about mental model bugs). SOLO has a good conceptual model and a good curriculum but the methodology was not geared towards quantifying the problems reported. PT501 is similar in this respect but has a weaker conceptual model and curriculum which lead to more interpretive problems. Open Logo has a potentially good conceptual model but the particular implementation has a number of severe problems including disastrous error messages, and the tutorial manual hovers between trying to teach Logo as a programming language and teaching Logo's educational philosophy, and is not successful at either.

One further general conclusion about the study then is that both the types of problems and the range of problems that students experience (and report) will be dependent on the conceptual model, the curriculum and the method of investigation. Jones and O'Shea (1979) report a number of barriers to the effective use of CAL at a distance, and proposed a Chinese box model. The Chinese box model applies here, too, as a model of barriers of a view of the novice's experiences, as shown in figure 10.1.

PT501's conceptual model could be improved, and this is the first barrier. Logo was better in this respect, but is "barred" by the second barrier, the curriculum. The conceptual model is dependent on the tutorial manual which carried it, and Logo doesn't do very well here. SOLO scores well on both counts but there is a final barrier of method, and the methods used in the SOLO study did not yield as much as the DESMOND study.



PT501

Figure 10.1: A "Chinese box" model of novices' experiences with the four languages

10.3 FUTURE WORK

In future studies concerned with the problems that novices encounter, the curriculum should be analysed carefully to see where various concepts are introduced (e.g. variable, flow of control, etc.) and most importantly whether these are tested by the exercises included in the instructional material. If they are not, they should be supplemented by additional questions inserted into the text. Subjects should also fill in a detailed comment or log book (as was done with

DESMOND). This will then give a good basis for analysing errors related to particular concepts. In addition, protocol or interview data should be collected, as error data cannot yield information on erroneous models and hypotheses. (Where possible, protocol data is preferable to interview data, as it is collected at the same time as the processes are occurring). It would not have been possible to understand the kinds of misconceptions that the PT501 students had without collecting detailed protocols.

Transfer studies need to address the above points too, and to ensure that the same concepts, although they are treated differently, are covered in both curricula. In order to study transfer, it is possible that a longer period of exposure is needed than was given in this study. Moreover, if we are interested in low level languages it should be noted that subjects often do not find such material motivating. Using instructional material such as that provided with DESMOND, and implementing the recommendations made, should overcome this problem. Finally, language independent tests should be used to assess competence and compare groups.

The analysis of the problems that the novices had led to a number of recommendations for changing the curricula, and the way languages are presented, i.e. the conceptual models. Whilst du Boulay, O'Shea and Monk (op. cit.) have provided good groundwork for the use of conceptual models in teaching novices, the guidelines that they suggest are not sufficient. One major improvement would be to ensure that wherever possible, a functional description of the model should be displayed in addition to the state descriptions and procedural descriptions. Two brief examples illustrate how this could be done for SOLO and DESMOND.

SOLO

In chapter 6, an example was given of a subject who said she didn't understand how to achieve "either/or". The subject knew the goal of the task she wanted to carry

out, but not how to achieve that goal. The WEAKASSESS procedure was given as an illustration of how SOLO can simulate a "weak" inference: that a person is fit if he or she does A or B or C. The STRONGASSESS procedure illustrated a "stronger" inference: a person is fit only if he or she does A and B and C. So together, the two procedures show how to achieve disjunction and conjunction. However, although the student's attention is drawn to the difference between the two procedures, and to the usage of CONTINUE and EXIT, the procedures are not explicitly labelled as ways of achieving an OR or an AND. Another problem is that with just one example, it is difficult to know which aspects to attend to. A functional description would give the goal and an algorithm for achieving it. In this case it could be provided by giving the STRONGASSESS and WEAKASSESS pocedures and showing how they represent "OR" and "AND" decisions. One way of doing this is to provide decision trees, as shown in figures 6.5a-d which are reproduced below and are annotated to show how the SOLO control structures can be used to achieve AND and OR.

DESMOND

The best way to give a functional description of the DESMOND conceptual model is to give the plans which were discussed in chapter 9. The instructional manual gives examples of some of these plans in the form of short programs, but does not give the plans. For example, the very first program given uses the state of switches. This involves two sub-goals: displaying the state of a device, and a looping plan (so that the display is updated if one of the switches is moved). The plan could be given as shown below:

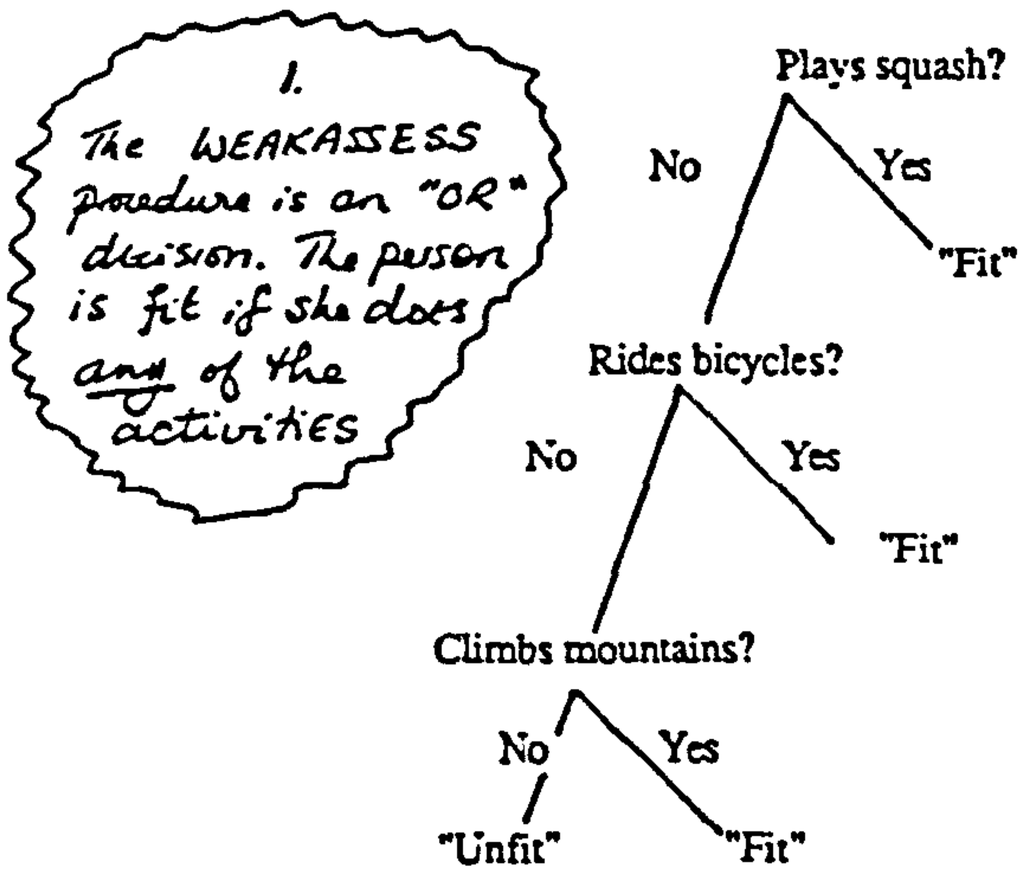


Fig 6.5a
Decision tree for WEAKASSESS

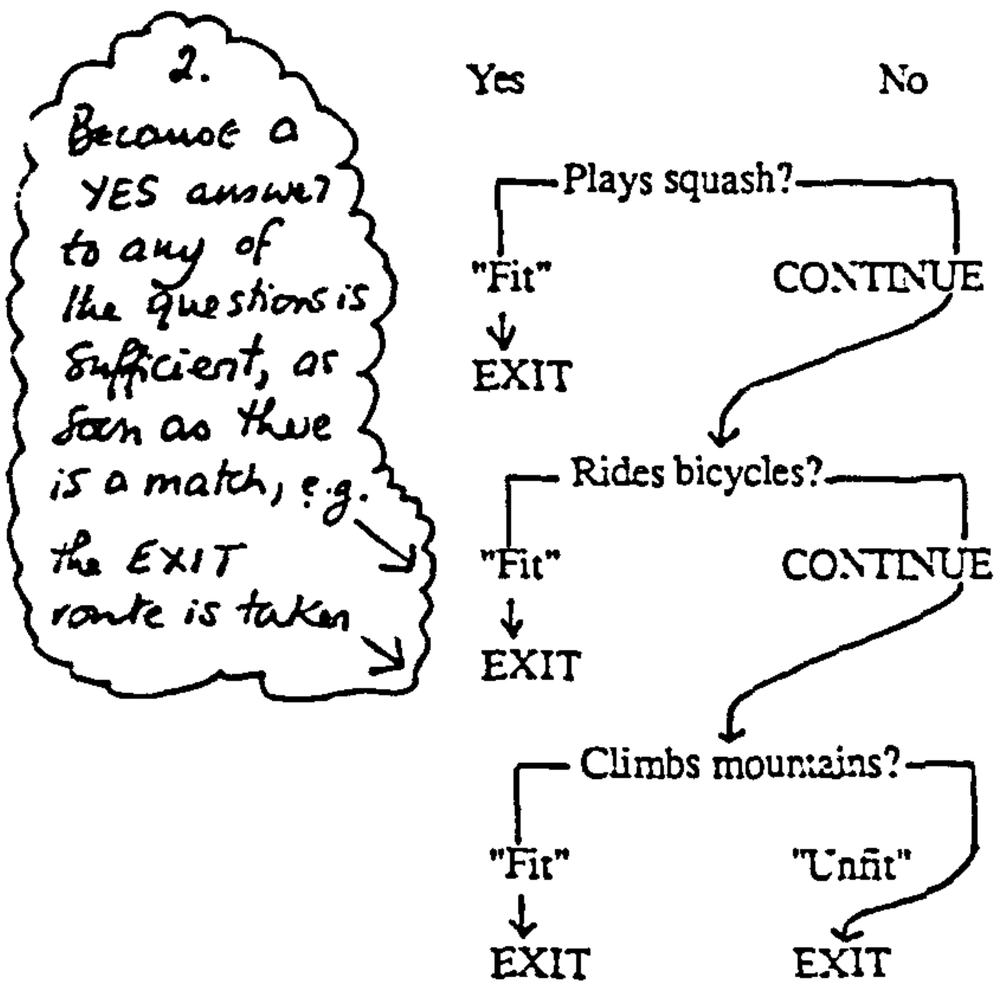


Fig 6.5b Chain diagram showing flow of control in WEAKASSESS

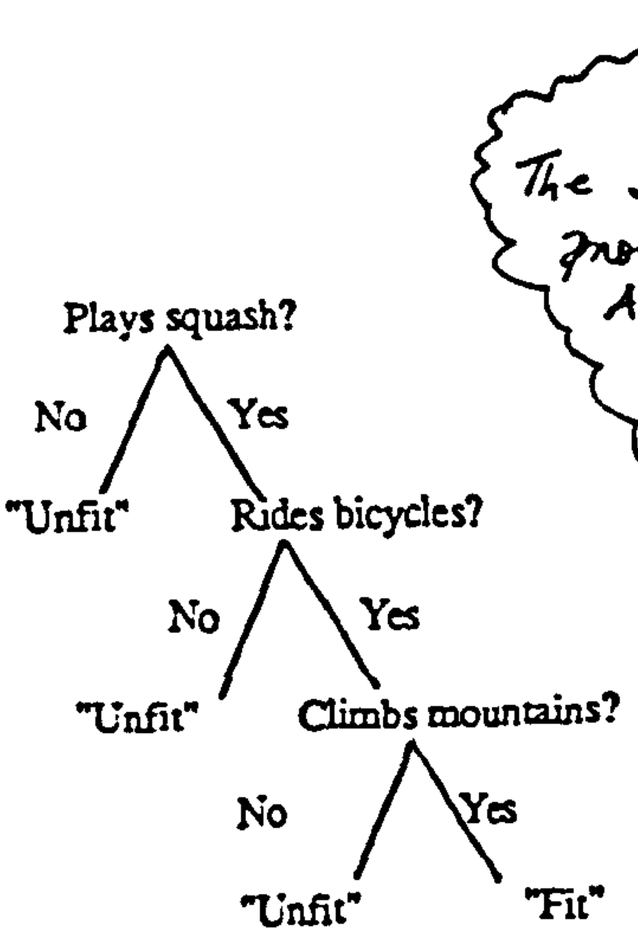


Fig 6.5c
Decision tree for STRONGASSESS

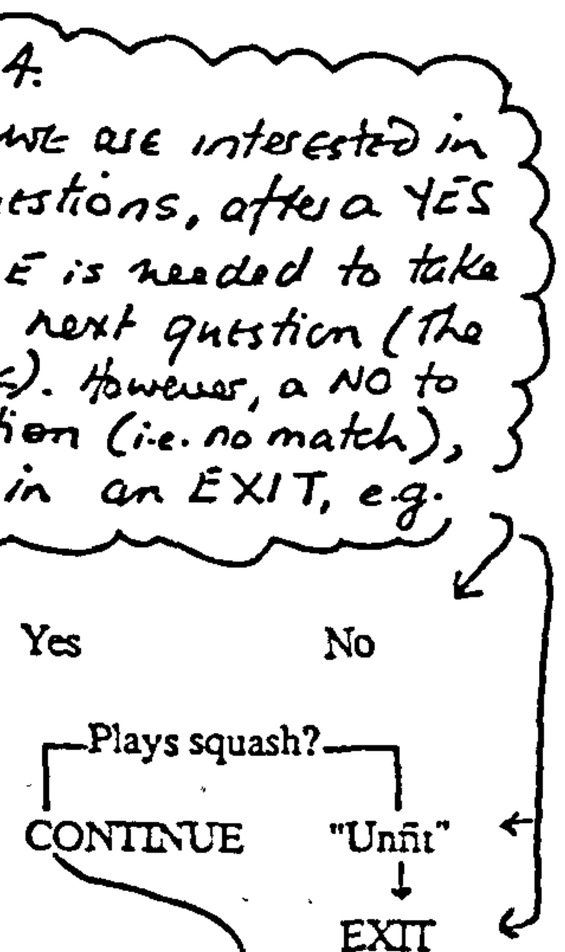


Fig 6.5d: Chain diagram showing flow of control for STRONGASSESS

Plan	Function	Outline
1	Display the state of a device	1 LDA [Address of device]
1a(Sub plan)	Display routine	2 JSR 200 or 205
1b(Sub plan)	Loop	3 JMP 000 [Repeat to update display]

It would be helpful to also show how the steps of the algorithm themselves achieve sub-goals and can therefore be seen as sub-plans:

ßub Plan	Function	Outline
1a	Display device	LDA [Address of device] JSR 200 or 205 [Jump to binary or denary display routine]
1b	Loop	JMP 000 [Repeat to update display]

Providing such a functional model, and giving the learners plans clearly effects how the curriculum is taught. A common strategy in the curricula studies was to ask the learner to discover a plan for themselves, as in Logo, or to work out the plan as part of writing a program for an exercise; but many learners were unable to do this. By giving learners a functional model, their task would be made much less demanding (such as asking them to combine two plans) and therefore they would be more likley to be successsful.

These recommendations should be implemented, and a new group of subjects be taught using the new curriculum, and their results compared to those of the subjects in this study. Questions such as: "Is curriculum A more effective than curriculum B?" are always problematic in educational research. In this instance however, given that error data exists, it should be possible to develop a new curriculum, which incorporates the recommendations made and which sets the

same exercises, so that a comparative study would be quite feasible. It is less feasible to carry out a comparative study using Open LOGO, as the changes needed to the curriculum are more substantial, but the analysis of the current Open LOGO tutorial manual has pointed to a number of areas that would benefit from improvement, and this would help future generations of novices.

Another area that needs further investigation is the issue of the possible conflict between the style of the distance learning materials used in this study and the programming domain. This was not originally an area of investigation but an issue that emerged from the study, and as such, the evidence is tentative. What is clear is that learners will be actively interpreting the instructional material that they have, and so will be misinterpreting and developing faulty models. Producing the "perfect" text, where learners automatically develop models which are the same as the conceptual models given, is not possible to do. The emphasis now should be on researching into ways of supporting students when they develop faulty models, and ways of helping them to work out where they have gone wrong, and put it right, for themselves.

References

Adelson, B. Problem Solving and the development of abstract categories in programming languages. Memory and Cognition, 9, 422-433, 1981

Allwood, C. M. and Eliasson, M. Analogy and other sources of difficulty in novices' very first text editing. International Journal of Man-Machine Studies, 27, 1-22, 1987

Anzai, Y. and Uesato, Y. Is Recursive Computation Difficult to Learn? Pittsburgh, PA: Psychology Department, Carnegie-Mellon University, 1982

Bayman, P. Effects of instructional procedure on learning a first programming language. Unpublished doctoral dissertation. University of California, Santa Barbara, 1983

Bonar, J. Natural Problem Solving Strategies and Programming Language Constructs, Proceedings of the Fourth Annual Conference of the Cognitive Science Society. Ann Arbor, 1982

Bott, R. A. A Study of Complex Learning: Theory and Methodologies, C.H.I.P. Report No 82, Dept. of Psychology, U.C.S.D. California, 1979

Breuker, J. Availability of Knowledge. C-O-W-O publications. Amsterdam, 1982

Brooks, R. Towards a theory of the cognitive processes in computer programming. International Journal of Man-Machine Studies, 9, (6), 737-752, 1977

Carroll, J. M. and Mack, R. L. Metaphor, Computing Systems, and Active Learning. Report RC9636, Computer Science Department, IBM Watson Research Centre, Yorktown Heights, N.Y. 1982

Carroll, J.M. and Thomas, J.C Metaphor and the Cognitive Representation of Computing Systems, IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-12, No 2, March-April, 1982

Carver, S. M. and Klahr, D. Assessing children's LOGO debugging skills with a formal model. Journal of Educational Computing Research, 2 (4), 487-525, 1986

Carver, S.M. LOGO debugging skills: Analysis, instruction, and assessement. PhD Dissertation. Department of Psychology, Carnegie-Mellon University, 1986

Carver, S.M and Risinger, S.C. Improving children's debugging skills. In Olson, G.M., Sheppard, S and Soloway, E. (Eds.) Empirical Studies of Programmers: second workshop, Ablex, 1987

Chase, W.G. and Simon, H.A. Perception in chess. Cognitive Psychology, 4, 55-81 , 1973

Conway , M. and Kahney, H. Transfer of Learning in Inference Problems: Learning to program recursive functions. In Hallam, J. and Mellish, C. (Eds.) Advances in Artificial Intelligence. Proceedings of the 1987 AISB conference. John Wiley and Sons, 1987

D'Arcy, J. Learning Pascal after BASIC. In Shackel, B. (Ed.) Human Computer Interaction - Interact '84 North Holland, 1985

Davies, S. The Nature and Development of Programming Plans. Internal Report from the Department of Computer Science and Mathematics, Huddersfield Polytechnic, 1988

Du Boulay, Some difficulties of Learning to Program. Journal of Educational Computing Research, Vol. 2 (1), 1986

Du Boulay, B. and O'Shea, T. Teaching Novices Programming. In Coombs, M. (ed.) Computing Skills and Adaptive Systems, Academic Press, 1980

Du Boulay, B., O'Shea, T. and Monk, J. The Glass Box inside the Black Box: Presenting Computing Concepts to Novices, International Journal of Man-Machine Studies, 14, 237-249, 1981

Easley, J.A. The structural paradigm in protocol analysis Journal of research in science teaching, 11, 281-290, 1974

Eisenstadt, M., SOLO: Units 3 &4, D303 Cognitive Psychology, Open University, 1978

Eisenstadt, M., A friendly Software Environment for Psychology Students. AISB Quarterly, 1989

Eisenstadt, M. and Lewis, M. Errors in an Interactive Programming Environment: Causes and Cures. HCRL report no 4, Psychology Department, Open University, (2nd edition) 1985

Ericsson, K. A. and Simon, H. Verbal Reports as Data. Psychological Review, 5, 230-248, 1981

Galotti, K. M. and Ganong, W.F. III What Non-Programmers know about Programming: Nat. Lang Proc. Spec. Int J Man-Mach Studies, 22, 1 - 10, 1985

Gentner, D. Structure-Mapping: A Theoretical Framework for Analogy, Cognitive Science, 7, 155-170, 1983

Gick, M.L. and Holyoak, K.J. Analogical problem solving. Cognitive Psychology, 12, 306-355, 1980

Gilbert, J. K. and Watts D. M. Concepts, misconcsptions: Changing perspectives in Science Education. Stud. Sci. Educ. 5 61-74, 1984

Gilmore, D. J. The perceptual cueing of the structure of computer programs. Unpublished PhD thesis, Psychology Department, University of Sheffield, 1986

Gilmore, D. J. and Green, T. R. G. The comprehensibility of programming notations in Human-Commputer interaction - INTERACT 84 (ed) B Shackel, North Holland, 1985

Glaser, B. G. and Strauss, A. L. The discovery of grounded theory: strategies for qualitative research, Weidenfeld and Nicolson, London, 1968

Green, T. R. G. Programming as a cognitive activity. In Smith, H.T and Green, T.R.G. (eds.) Human Interaction with Computers. London, Academic Press, 1980

Green, T. R. G. and Arblaster, A.T. As you'd like it: contributions to easier computing, Memo No 373, MRC Social and Applied Psychology Unit, University of Sheffield, U.K. 1980

Green, T. R. G., Sime. M. E. and Fitter, M.J. The Art of Notation. In Coombs, M. and Alty, J. (Eds.) Computing Skills and Adaptive Systems. Academic Press, 1980a

Green, T. R. G., Sime, M.E. and Fitter, M.J. The problems the programmer faces, Ergonomics, 1980b

Green, T. R. G., Bellamy, R. K. E. and Parker, M. Parsing and Gnisrap: A Model of Device Use. In Olson. G.M., Sheppard, S. and Soloway, E. (Eds) Empirical Studies of programmers: second workshop. Ablex, New Jersey, 1987

Green, T. R. G. and Cornah, A. J. The Programmer's Torch. In Shackel, B, (Ed.) Human-Commputer Interaction -INTERACT 84. North Holland, 1985

Harvey, B. Why Logo? In Yazdani, M (Ed). New Horizons in Educational Computing, Ellis Horwood, 1984

Hasemer, T An empirically based debugging system for novice programmers. HCRL report no 6, Dept of Psychology, Open University, 1983

Heller, R. S. Different Logo Teaching Styles: Do They Really Matter? In Soloway, E. and Iyengar, S. (Eds.) Empirical Studies of Programmers. Ablex, N.J. 1986

Hofstadter, D. Godel, Escher and Bach: The Eternal Golden Braid. Harvester Press, 1980

Holyoak, K.J. and Thagard, P. R. A Computational Model of Analogical Problem Solving, In Vosniadou and Ortony, A. (Eds.) Similarity and Analogical Reasoning, Cambridge University Press, 1989

Howe, J. A. M., O'Shea, T. and Plane, F. Learning through LOGO: Outlining the Non-Reactive Evaluation Study, DAI Working Paper 40, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, 1978

Hoyles, C and Noss, R. Synthesising Mathematical Conceptions and their Formalisations through the Construction of a LOGO-Based School Mathematics Curriculum, Department of Mathematics, Statistics and Computing, University of London Institute of Education, December 1985.

Jeffries, R. A comparison of the debugging behaviour of expert and novice programmers. Paper presented at AERA annual meeting, March 1982

Johnson, W. L., Draper, S. and Soloway, E. An Effective Bug Classification Scheme Must Take the Programmer into Account. In The Proceedings of the Workshop on High-Level Debugging, Palo Alto, 1983

Jones, A. Issues in post awareness teacher training: a case study of the Micros in Schools project, In proceedings of "Computers in Education, Expansion or Contraction?", Research Group OW and OC, University of Utrecht, 1986

Jones, A. and Preece, J. Training teachers to assess computer software. In Computer Education, November 1984

Jones, A., and O'Shea, T. Barriers to the Use of Computer Assisted Learning in British Journal of Educational Technology, 3,13, 207-217, 1982

Jones, A., and Scanlon, E Protocols and production systems, CAL Research Group Technical Report No 32, 1983

Jones, A., Scanlon, E and O'Shea, T. (Eds) The Computer Revolution in Education: New Technologies for Distance Teaching. The Harvester Press, Sussex, 1987

Kahney, H. An in-depth study of the cognitive behaviour of novice programmers, HCRL Technical report no 5, December 1982, Open University, Milton Keynes.

Kelly G. A. The Psychology of Personal Constructs, Vol. 1,2 Norton, New York, 1955

Kirkup, G. M205 End of Year Report., CITE Report No 69, Institute of Educational Technology, Open University, 1988

Kurland, D. M., and Pea, R.D. Children's mental models of recursive Logo programs, Journal of Educational Computing Research, Vol 1 (2), 1985

Kurland, D. M., Pea, R.D., Clement, C. and Mawby, R. A Study of the development of programming abilty and thinking ability in high school students. Journal of Educational Computing Research, Vol 2 (4), 1986

Kurland, D. M., Clement, C., Mawby, R. and Pea, R.D. Mapping the cognitive demands of learning to program, in Kurland, D. M. (ed) AERA Symposium: Developmental Studies of Computer Programming Skills, Technical Report No. 29, Bank Street College of Education, N. York, 1984

Lakoff, G. and Johnson, M. Metaphors we live by. Chicago: Chicago University Press, 1980

Lashley, K.S. The behaviouristic interpretation of consciousness 11, Psychological Review, 30 329-353, 1923

Lewis, M. Improving SOLO's user-interface: An empirical study of user behaviour and a proposal for cost effective enhancments to SOLO. CAL Research Group Report No 7, Open University, 1980

Lewis, C. and Mack, R. The role of abduction in learning to use a computer system. Research report RC9433, No 41620, IBM Watson Research Centre, Yorktown Heights, N.Y. 1982.

Linn, M.C. The Cognitive Consequences of Programming Instruction in Classroms Educational Researcher, 14, 5 - 29, 1985

Mack, R., Lewis, C. and Carroll, J. Learning to use Word Processors: Problems and Prospects, IBM Watson Research Centre Report C9712, No 42886, Yorktwon Heights, N.Y. 1982

Mayer, R. E. Different problem-solving contingencies established in learning programming with and without a meaningful model. Journal of Educational Psychology, 67, 725-734, 1975

Mayer, R. Some Conditions of Meaningful Learning for Computer Programming: Advance Organisers and Subject Control of Frame Order, Journal of Educational Psychology, 68 2., 143-150, 1976

Mayer, R .E. Advance Organizers that Compensate for the Organisation of Text. Journal of Educational Psychology, Vol 70, pp 880-886, 1978

Mayer, R. A Psychology of Learning BASIC. Communications of the ACM, 22, 11, 589-593, 1979b

Mayer, R. E. Can advance organisers influence meaningful learning? Review of Educational Research, 49, 371-383, 1979a

Mayer, R. E. Elaboration Techniques for Technical Text: An Experimental Test of the Learning Strategy Hypothesis. Journal of Educational Psychology, 72, 770-784, 1980

Mayer, R. E. Cognitive Aspects of Learning and Using a Programming Language. In Carroll, J. M. (Ed.) Interfacing Thought: Cognitive Aspects of Human-Computer Interaction. MIT Press, Cambridge, Mass. 1987

Mayer, R. and Bayman, P. Psychology of Calculator Languages: A Framework for Describing Differences in Users' Knowledge. Communications of the ACM, 24, 8, 511-520, 1981

McArthur, C. D. Emas LOGO: user's guide and reference manual. D.A.I. Occasional Paper No. 1. Department of Artificial Intelligence, Edinburgh University, 1974

McKeithen, K. B., Reitman, J. S., Reuter, H.H. and Hirtle, S.C. Knowledge organisation and skill differences in computer programmers. Cognitive Psychology, 1981, 3, 307-325

Miller, L. A. Naive programmer problmes with specification of transfer-of-control. AFIPS National Computer conference, 1975, 44

Miller, L. A. Natural Language Programming: Styles, Strategies and Contrasts. IBM systems Journal, 20, 184-216, 1981

Newell A. On the analysis of human problem solving protocols. In Johnson-Laird, P. and Wason, P (Eds.) Thinking: Readings in Cognitive Science. Cambridge University Press, 1977

Nisbet, R. E and Wilson, T.D. Telling more than we can know: Verbal reports on mental processes. Psychological Review, 77, 231-259, 1987

Norman, D. A. What Goes On in the Mind of the Learner? in W. J. McKeachie, (Ed), New Directions in Learning and Teaching, Jossey-Bass, San Fransisco, 1979

Norman, D. A. Some Observations of Mental Models in Mental Models, ed Albert Stevens and Dedre Gentner, Lawrence Erlbaum Associates, New Jersey, 1983

Noss, R. How do children learn mathematics with LOGO? Journal of Computer Assisted Learning, 3 (1), 2-12, 1987

Novak, J. D. and Gowin, D. B. Learning How to Learn. Cambridge University Press 1984

O'Malley, C., Draper, S. and Riley, M. Constructive interaction: A method for studying human-computer interaction. In Shackel, B (ed) Human-Computer Interaction - Interact '84. Amsterdam: North Holland, 1985

Open University. Microcomputers for Managers: A Short Pack, 1979

Open University. Inside Microcomputers: A Short Pack, 1984

Open University. Learning About Microcoelectronics: A Short Pack, 1984

O'Shea T. A self-improving quadratic tutor. International Journal of Man-Machine Studies, 11, 97-124, 1979

O'Shea, T. The Open University 'Micros in Schools' project. In Lovis, F. B., and Tagg, E. D. (Eds.) Informatics and teacher training. North Holland, 1984

O'Shea, T. and Self, J. Learning and teaching with Computers: Artificial Intelligence in Education, Harvester Press, 1983

O'Shea, T., Young, R., Evertz, R., Jones, A., Mellis, W., Scanlon, E., Floyd, A. and Kahney, H. Applicability and limits of production rule models: ten case studies. Paper given at Workshop on Modelling Cognition, Lancaster University, July, 1985

Papert, S. Mindstorms Harvester Press, 1980

Pea, R. and Kurland, M. The Cognitive Effects of Learning Computer Programming, New Ideas in Psychology, Vol. 2. pp 137-168, 1984

Pea, R.D. Language-Independent Conceptual Bugs in Program Understanding, Journal of Educational Computing Research, 2 (1), 25-36, 1986

Perkins, D. M., Hancock, C., Hobbs, R., Martin, F. and Simmons, R. Conditions of Learning in Novice Programmers. Journal of Educational Computing Research, 2,37- 55, 1986

Perkins, D. M., and Martin, F. Fragile Knowledge and Neglected Strategies in Novice Programmers. In Soloway, E., and Iyengar, S. (Eds) Empirical Studies of programmers, N.J., Ablex, pp 213-229, 1986

Piaget, J. Development and Learning, in R.E, Ripple and V.N. Rockcastle, (eds) Piaget Rediscovered, Ithaca N;Y; Cornell University School of Education, 1964

Rist, R. S. Program Plans and the Development of Expertise, Department of Computer Science, Yale University, 1985

Rist, R. S. Plans in Programming: Definition, Demonstration and Development, in Empirical Studies of Programmers, (eds.) Soloway, E. and Iyengar, S, Ablex, N.J. 1986

Rumelhart, D. E. and Norman, D. A. Representation in Memory In Atkinson, R. C.,Herrnstein, R. J., Lindzey, G. and Luce, R. D. (Eds.) Handbook of Experimental Psychology, Wiley and Sons, 1983

Sanders, R. S. Classroom Questions. Jersey Bros., 1961

Scanlon, E. and O'Shea, T. How novices solve physics problems. In Jones, A., Scanlon, E. and O'Shea, T. The Computer Revolution in Education, (Eds.) Harvester Press, 1987

Scanlon, E., Hawkridge, C., and O'Shea, T. Modelling problem solving: scripts and production systems. CAL Research Group report no 36, Open University, 1983

Schank, R. and Abelson, R. Scripts, Plans, Goals and Understanding, Hillsdale, N.Jersey, Lawrence Erlbaum Associates, 1977

Schank, R. Dynamic Memory: A Theory of Learning in Computers and People. Cambridge University Press, Cambridge Mass. 1982

Sharples, M. Phrasebooks and boxes: microworlds for language. In Duncan, K., and Harris, D. (Eds.) Computers in Education. Elsevier Science, 1985

Shneiderman, B. Exploratory experiments in programmer behavior. International Journal of Computer and Information Science, 5, 124-143, 1976

Shneiderman, B. and Mayer, R.E (1979) Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 1979, 7, 219-239. Reprinted in Curtis, B. (ed) Human Factors in Software Development, IEEE EHO, 186-189, 1981

Soloway, E. From Problems to Programs via Plans: The content and structure of Knowledge for Introductory LISP Programming. Journal of Educational Computing Research, 1, 157-172, 1985

Soloway, E., Bonar, B. and Ehrlich, K. Cognitive Strategies and Looping Constructs: An Empirical Study. Communications of the ACM, 26, 853-860, 1983

Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J. What do novices know about programming? In Shneiderman, B. and Badre, A. (Eds.) Directions in Human-Computer Interactions. Ablex, N.J. 1982

Soloway, E. and Ehrlich, K. Empirical studies of Programming Knowledge. In IEEE Transactions on Software Engineering, September, 1984

Taylor, J. Programming in Prolog: An in-depth study of problems for beginners learning to program in Prolog. D.Phil thesis, School of Cognitive Sctudies, Sussex University, 1987

Tennyson, R.D. and Rasch, M. Linking cognitive learning theory to instructional prescriptions. Instructional Science, 17, 369-386, 1985

Vessey, I. On matching programmers' chunks with program structures: An empirical investigation. International Journal of Man-Machine Studies, 27, 65-89, 1987

Weidenbeck, S. Beacons in computer program comprehension. International Journal of Man-Machine Studies, 25, 697-709, 1986

Weidenbeck, S. Learning iteration and recursion from examples. International Journal of Man-Machine Studies, 30, 1-22, 1989

Widowski, D. and Eyferth, K. Representation of computer programs in memory: comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In ECCE 3: Third International Conference on Cognitive Engineering, Paris, INRIA, Paris, 1986

Weyer, S. A. and Cannara, A.B. Children learning computer programming: experiments with languages, curricula and programmable devices. Technical report no. 250, Institute for mathematical studies in the social sciences, Stanford University, Stanford, California, Jan 1975

White, R. H. Effects of Pascal upon the learning of Prolog - a preliminary study. Unpublished paper, University of Edinburgh, 1988

Young, R. The Machine Inside the Machine: Users' Models of Pocket Calculators International Journal of Man-Machine Studies, 15, 1, 51-85, 1981

Appendix 4

Appendix 4.1: Open Logo instructions

Appendix 4.2: The PT501 instruction set

Appendix 4.3: DESMOND's instruction set

Appendix 4.1: Open LOGO instructions

Instruction	Effect
PRINT '	Prints the contents of quoted string
PRINT [list]	Prints the contents of a list
PRINT (variable)	Prints variable value
MODE (1 to 8)	Selects one of 8 modes
DRAWING	Selects DRAWING mode
FORWARD x	Moves the turtle x units in the direction in which turtle is facing, and (default) leaves trace on screen
RIGHT x	Turns turtle position x degrees right (i.e. clockwise)
LEFT x	Turns turtle position x degrees left (i.e. anti clockwise)
PENUP	"Lifts pen", i.e. subsequent turtle moves do not leave trace
PENDOWN	Restores "pen" position i.e. subsequent turtle moves do leave trace
BACKWARD x	Moves turtle backwards x units, - i.e. in the opposite direction to which the arrowhead is facing
BLANK	Wipes the screen - including graphics and text
REPEAT x[ ]	Carries out the instruction in the brackets x times
CLEAN	Removes graphics from graphics window

CENTRE	Puts the turtle in the centre of the graphics screen
FENCE	Creates a border to the graphics window
RUBBER	Subsequent drawing commands rub out lines created by previous commands
WRAP	'Wraps' turtle around graphics window, i.e. when it leaves graphics window it then immediately reappears on the opposite side of the screen
WINDOW	Undoes wrapping command
TEXT	Removes graphics window (if exists) and any text typed in then scrolls to fill up the whole screen
ARCL x y	Draws curved line (anti-clockwise) of radius x. If y = 360 it will be full circle If y = 180 it will be half circle, etc
ARCR x y	Draws curved line (clockwise) of radius x. If y = 360 it will be full circle If y = 180 it will be half circle, etc
DRIVE	Puts turtle in immediate graphics mode where turtle is moved by pressing certain keys (e.g. arrows to move up and down, F to move faster, S to slow down)
TO x(variable)	Creates procedure x

Appendix 4.2: The PT501 instruction set

Instruction	Form taken	Effect
Data moving instructions		
LOAD	Ld xxx	Enters binary equivalent of xxx into accumulator, replacing previous contents
LOAD FROM REGISTER	Ld ry	Copies contents of register with address ry into accumulator, replacing previous contents .
ADD FROM REGISTER	Ad ry	Adds the contents of register with address ry to contents of accumulator and leaves result in accumulator
DEC	DEC	Decreases the contents of accumulator by one.
Logical instructions		
EXCLUSIVE-OR	Eo *zzzzzzzz*	Performs an EXCLUSIVE-OR operation between the contents of the accumulator and *zzzzzzzz*, leaving the results in accumulator
EXCLUSIVE-OR WITH REGISTER	Eo ry	Performs an EXCLUSIVE-OR operation between the contents of the accumulator and the contents of the register with address ry leaving the result in accumulator
Control instructions		
JUMP	Ju xxx	Puts xxx into the program counter causing the microcomputer to jump to the location whose address is xxx
JUMP IF ZERO	JIFO	Tests the contents of accumulator. If zero, puts xxx into the program

		counter, otherwise the program counter is increased normally
CALL	CALL xxx	Puts onto the stack the return address and then puts xxx into the program counter. (Uses the contents of the register with address r6 as the stack pointer; and increases the stack pointer.)
Input/Output instructions		
OUT	Out	Interprets the value of the accumulator as on and off values for the traffic lights.
IN In		Inspects the state of the blank key on the data keypad. Puts 255 in the accumulator if the key is not being pressed, and 0 if it is.
EXCHANGE WITH REGISTER	144 Ry	Interchanges the contents of the accumulator and specified register
RETURN	240	Copies from the stack the return address into the program counter (after having first decreased the stack pointer's value by one).

Appendices 5

Appendix 4.3: DESMOND's instruction set

Instruction	How it's used	Effect
NOP	NOP	No operation
Data-moving instructions		
LDA	LDA xxx	Load Accumulator from memory location xxx
STA	STA xxx	Store copy of Accumulator in memory location xxx
LDI	LDI xxx	Load Accumulator with xxx
Data-manipulating instructions		
ADD	ADD xxx	Add to Acc from mem location xxx
ADI	ADI xxx	Add to Accumulator xxx
DEC	DEC	Decrement 1 from accumulator
AND	AND xxx	Perform logical AND on Acc with contents of memory location xxx
NOT	NOT	Invert the accumulator
SHR	SHR	Shift right one bit
Program control instructions		
JMP	JMP	Unconditional jump
JSR	JSR	Jump to subroutine
RET	RET	Return from subroutine
CMP	CMP xxx	Compare Accumulator with contents of memory location xxx
JZ	JZ xxx	Jump to mem location xxx if zero flag set to 1

JLO	JLO xxx	Jump to mem loc xxx if lower flag set to 1

Appendix 6

Appendix 6.1: The SOLO summer school project

Appendix 6.2: Some examples of the exercises in the SOLO manual

Appendix 6.1: The SOLO summer school project

A1: ARTIFICIAL INTELLIGENCE

SOLO and its applications in cognitive psychology

We hope that by now you will be convinced of the value of artificial intelligence (AI) in the study of cognitive psychology. In Units 3–4 you were introduced to a programming language called SOLO. At that time you were shown a few examples of the types of programs that can be written with SOLO. Later units emphasized the relevance of an AI approach to problems in such areas as perception and memory.

The aim of the AI trailer is to give you a mid-course refresher on the ideas developed in Units 3–4, and to give you the chance to develop your AI skills a little further. The project will give you the opportunity to apply these in the light of material you have come across since having done TMA 02; or perhaps even to preview some of the material which you will be studying just after Summer School (e.g. Block 4 on problem solving).

You will have the chance to write simple AI programs in the areas of memory, perception, learning and problem solving, capitalizing on all the things you now know about these areas which you did not know when you originally worked through Units 3–4. For example, for memory, you have now read about Collins and Quillian and how working memory is used in mental arithmetic tasks; in perception you have read about cues and schemas; in the units on learning and instruction you are currently being introduced to the idea of schemas and prototype descriptions. Each of the areas provides a potential topic for you to work on in the course of the AI project.

In many cases this will involve making use of the extensions to SOLO described in the Appendix of Units 3–4, and therefore discovering some of the capabilities of SOLO of which you were not previously aware. The extent to which you use these extensions will depend on the topic you choose and your own personal bias. You have probably already discovered that programming is a personal thing!

The AI trailer

This session will enable you to learn how to use the computing equipment, and will introduce the four main topic areas. For each topic a program will already have been written to illustrate a working solution for that topic. In this short trailer session you will be working in small groups around a microcomputer trying out each of the four programs. At this stage you will not be required to write any programs, only type in the odd word or two. Even if you do not go on to do the AI project itself, we hope this trailer will give you some idea of how SOLO can be used to study a wide range of problems in cognitive psychology.

The AI project

You will be working in groups of four, attempting to develop a program to tackle one of the topics listed below. Your group may not be satisfied with just one program, and you may therefore make more than one attack on one topic, or

even attempt more than one topic. Whatever you decide to do, try to make it a group effort, and aim to produce complete programs and not a series of attempts.

From the trailer session you will have seen one possible solution to the topic you have decided on, but this will by no means be a unique solution. You should *not* try to write a program that mimics the trailer; instead write a program that embodies the group's ideas about the topic. The appeal of AI programming is in its potential for examining your own ideas.

The topics have been selected to provide a challenge both to the student still mastering the art of programming, and also to the student wanting to push back the frontiers of AI. SOLO offers both simplicity and scope for practically anyone.

Topic 1 Propagating inferences

Sometimes an inference which we make may have further ramifications. For instance, when the Watergate burglars were caught, it was inferred that someone else behind the scenes was guilty. The further ramifications led to Nixon's resignation. A very simple inference, such as 'if someone is found to be guilty, then whoever that person works for is also guilty', can be represented using SOLO procedures. This topic asks you to implement this type of 'propagating inference' in SOLO.

Relevant reading Units 3–4, section 7 and section A.1.

Topic 2 Collins and Quillian

In Units 18–19, section 5.1, you read about a model developed by Collins and Quillian to account for the way people answer questions like 'Does a canary have wings?'. This topic asks you to implement a simplified version of the Collins and Quillian model, using SOLO notation. For instance, you will have to design a procedure called CONFIRM, so that when you type in

SOLO: CONFIRM CANARY HAS WINGS

then SOLO will respond either YES or NO, depending upon the structures stored in its data base. As you can imagine, the secret will be to construct a procedure which continues searching for relevant clues even when it finds that the triple

CANARY – – – HAS ——→ WINGS

is absent from the data base (in this case, relevant triples such as

BIRD – – – HAS – –→ WINGS

might very well be *present* in the data base).

You may also be interested in seeing how far your program can *simulate* the experimental results found by Collins and Quillian, e.g. the times taken to make 'false' responses.

Topic 3 Children's arithmetic skills

Why do children (and many adults) make errors when performing addition and subtraction? The underlying mental processes involved are more complicated than may be apparent at first glance. Since SOLO contains no special procedures for dealing with numbers, it puts you in an excellent position to develop *from scratch* a model of how children perform arithmetic tasks. The computer won't be able to 'cheat' and print out the mathematically correct result automatically, because it doesn't know how to! It will be *your* responsibility to design the proce-

dures which perform the various arithmetical operations. You will have a chance to see some data which describes mistakes commonly made by children, and systematically to modify your own programs to try to account for the occurrence of these mistakes.

Relevant reading Unit 15, section 4.1.

Topic 4 Schema matching

The idea of 'schemas' and 'structural descriptions' underlying perceptual processing was emphasized in Units 7 and 8. You already know how to use SOLO to create a structural description, although you haven't yet done so; the technique is precisely the same as that for building 'propositions' (Units 3–4, section 8). This topic asks you to create a SOLO procedure (or several procedures) which can *compare* two structural descriptions to see whether or not they are the same.

This comparison process is fundamental, since it may very well underlie elementary perceptual and learning skills. For instance, suppose you are comparing two descriptions, one of a pre-stored schema (say, for what an 'arch' looks like), the other of some unknown entity (say, some object which you are currently looking at and trying to identify). Then this comparison or 'matching' process will allow you to *recognize* the unknown object as an 'arch', if the two descriptions happen to match. Or, if you are told that the unknown object is an 'arch', then you may update your 'arch schema' (your pre-stored structural description of what an arch looks like) to take into account any new features possessed by this previously unknown object. The procedures you create will be simple ones designed to compare two small semantic network structures in order to see whether they are identical and to report back (i.e. print out) any differences if the structures turn out not to be identical.

Relevant reading Unit 7, section 4; Unit 8, section 2; Reading 12 by Winston in Johnson-Laird and Wason; Boden, pp 253–67.

TV 3, *Cues and schemas,* is also relevant.

SOLO does have some limitations, so that any venture outside this set of topics will require very careful consultation with your tutor. Even for some of the above topics you may require considerable advice and suggestions from your tutor to make your programming job easier. Sketch out a plan of your program before you begin. You may decide to divide the problem up amongst the group to start with, but this will require close cooperation to produce a final program that works properly. If you wish to see the related TV programmes (TV 1 to 4) before you start, this can be arranged by your tutor.

Useful references

BADDELEY, A.D. (1976) *The psychology of memory,* Harper and Row (pp 325–33).

BODEN, M. (1977) *Artificial intelligence and natural man,* Harvester Press (pp 253–67).

COLLINS, A.M. and QUILLIAN, M.R. (1969) 'Retrieval time from semantic memory', *Journal of Verbal Learning and Verbal Behaviour,* vol. 8, pp 240–47 (Offprint 5 in Offprints Booklet).

Appendix 6.2: Some examples of the exercises in the SOLO manual

SAQ 1 YOUR ANSWER (OUR ANSWERS TO SAQS ON p 94)

The notion of one-way relations has important psychological implications. Try to name everyone you know who wears glasses (only spend about 2 minutes trying), then answer the following questions:

(a) What, offhand, would be two different strategies for accomplishing this task?

(b) How might your knowledge of who wears glasses be represented in SOLO's data base (i.e. what relation name would you use to label the arrows, and which way would the arrows point)?

(c) What is the relevance of SOLO's one-way relations to the way in which people might search through their memories?

SAQ 3 YOUR ANSWER

Here's an example involving nonsense words, to test your understanding of the principles involved. If SOLO's data base contained the following description

```
FOO
 |
 |---BAZ--->GORT
 |
 |---XLIB--->RIJK
```

what would be the sequence of things we would have to type in so that SOLO's final description looked like this:

```
FOO
 |
 |---BAZ--->GORT
 |
 |---XLIB--->RIJK
 |
 |---LOON--->FUNG
 |
 |---MXLPLTZ--->GLURTO
```

SAQ 9

YOUR ANSWER

Here's a sample interaction with SOLO which includes the definition of a new version of the JUDGE procedure. Decide what SOLO's responses would be to lines (a),(b) and (c) below:

```
SOLO: TO JUDGE /X/
 ...:1  PRINT "HERE'S MY OPINION OF " /X/
 ...:2  CHECK /X/---ISA--->DOG
 .....2A IF PRESENT: PRINT /X/ "IS A FINE DOG"; EXIT
 .....2B IF  ABSENT: CONTINUE
 ...:3  CHECK /X/---VOTES--->  INDEPENDENT
 .....3A IF PRESENT: PRINT /X/ "IS VERY THOUGHTFUL"; CONTINUE
 .....3B IF  ABSENT: PRINT /X/ "IS ALL RIGHT"; CONTINUE
 ...:4  PRINT "AND THAT'S ALL I HAVE TO SAY"
 ...:DONE
OK...I NOW KNOW HOW TO 'JUDGE' /X/

SOLO: DESCRIBE MARY
MARY
 |
 |---ISA--->WOMAN
 |
 |---VOTES--->INDEPENDENT

SOLO: DESCRIBE JOHN
JOHN
 |
 |---ISA--->MAN
 |
 |---OWNS--->FIDO

SOLO: DESCRIBE FIDO
FIDO
 |
 |---ISA--->DOG
 |
 |---CHASES--->CATS
```

Now, how would SOLO respond to each of the following:

(a) SOLO: JUDGE FIDO
(b) SOLO: JUDGE JOHN
(c) SOLO: JUDGE MARY

In determining your answers be sure to go through the JUDGE procedure step by step, and pay careful attention to the control-statements!!

```
SOLO: TO WEAKASSESS /X/
 ...:1  CHECK /X/ ---PLAYS---> SQUASH
 .....1A IF PRESENT: PRINT "FIT"; EXIT
 .....1B IF  ABSENT: CONTINUE
 ...:2  CHECK /X/ ---RIDES---> BICYCLES
 .....2A IF PRESENT: PRINT "FIT"; EXIT
 .....2B IF  ABSENT: CONTINUE
 ...:3  CHECK /X/ ---CLIMBS---> MOUNTAINS
 .....3A IF PRESENT: PRINT "FIT"; EXIT
 .....3B IF  ABSENT: CONTINUE
 ...:4  PRINT "UNFIT"
 ...:DONE
OK, I NOW KNOW HOW TO 'WEAKASSESS' /X/
SOLO: TO STRICTASSESS /X/
 ...:1  CHECK /X/ ---PLAYS---> SQUASH
 .....1A IF PRESENT: CONTINUE
 .....1B IF  ABSENT: PRINT "UNFIT"; EXIT
 ...:2  CHECK /X/ ---RIDES---> BICYCLES
 .....2A IF PRESENT: CONTINUE
 .....2B IF  ABSENT: PRINT "UNFIT"; EXIT
 ...:3  CHECK /X/ ---CLIMBS---> MOUNTAINS
 .....3A IF PRESENT: CONTINUE
 .....3B IF  ABSENT: PRINT "UNFIT"; EXIT
 ...:4  PRINT "FIT"
 ...:DONE
OK, I NOW KNOW HOW TO 'STRICTASSESS' /X/
```

Suppose that SOLO's data base contained the following descriptions:

```
FRED
 |
 |---ISA---> MAN
 |
 |---PLAYS---> SQUASH
 |
 |---RIDES---> BICYCLES
 |
 |---CLIMBS---> MOUNTAINS
MARY
 |
 |---ISA---> WOMAN
 |
 |---PLAYS---> SQUASH
 |
 |---RIDES---> BICYCLES
```

How would SOLO respond to each of the following (be sure to work through the procedures step by step, and follow the control-statements *precisely*):

(a) SOLO: WEAKASSESS FRED
(b) SOLO: WEAKASSESS MARY
(c) SOLO: STRICTASSESS FRED
(d) SOLO: STRICTASSESS MARY

Suppose that SOLO's data base contained the following four dcescriptions:

```
FRED
  |
  |---ISA---> MAN
  |
  |---VOTES---> INDEPENDENT
MARY
  |
  |---ISA---> WOMAN
  |
  |---VOTES---> INDEPENDENT
JOHN
  |
  |---ISA---> MAN
JOAN
  |
  |---ISA---> WOMAN
```

How would you define a new version of the JUDGE procedure so that it behaved in the following way:

```
SOLO: JUDGE FRED
HERE'S WHAT I THINK OF FRED
I REALLY RESPECT FRED
HE IS AN INDEPENDENT THINKER
AND THAT'S ALL I HAVE TO SAY

SOLO: JUDGE MARY
HERE'S WHAT I THINK OF MARY
I REALLY RESPECT MARY
SHE IS AN INDEPENDENT THINKER.
AND THAT'S ALL I HAVE TO SAY

SOLO: JUDGE JOHN
HERE'S WHAT I THINK OF JOHN
HE CAN'T THINK FOR HIMSELF
AND THAT'S ALL I HAVE TO SAY
```

SAQ 11

YOUR ANSWER

Suppose SOLO's data base contained the following description:

```
SUSAN
  |
  |---ISA---> WOMAN
  |
  |---LIKES---> CARPENTRY
```

How would you redefine SUSS so that if you typed in

SOLO: SUSS SUSAN

SOLO would respond

I KNOW SUSAN IS FOND OF CARPENTRY

Appendix 7

Appendix 7.1: Initial Questionnaire

Appendix 7.2: Final Questionnaire

Appendix 7.1: Initial Questionnaire

1. Please write in your student serial number: [redacted] (16-23)

At several places on this form, codes (1, 2 etc.) are provided for answering (e. g. Yes 1, No 2). Please select your answer (e. g. 'Yes'), then draw a ring around the corresponding code, not around the word (e. g. Yes (1), No 2).

Please ignore bracketed numbers throughout. E. g. (24). Do not mark or ring the bracketed numbers.

2. Which part of D303 are you currently studying? *(write in)*

Methodology Handbook (Unit 5)
& doing TMA 01

3. Please list below any Open University courses you intend to study in the future, and indicate by ringing code 1, 2 or 3 how likely you are to study each one you list.

Title of Course/Course Code *(write in)*	Might take...	Quite likely to take	Highly likely to take	
Personality + Learning E201	1	2	(3)	(24, 25, 26)
Cog. development – [redacted] E362	1	2	(3)	(27, 28, 29)
Modern art A351	1	2	(3)	(30, 31, 32)
	1	2	3	(33, 34, 35)
	1	2	3	(36, 37, 38)
	1	2	3	(39, 40, 41)

4. Please list below any courses you have studied other than OU courses

Title of course or subject area	*Please tick in whichever of these columns best describes the 'level'* GCE level 'O'	GCE level 'A'	Undergraduate level or above	Other (Please specify)
English Literature. Religious Knowledge	✓	A ✓		
General Knowledge. Physiol. Anatomy + hygeine.	"	OA ✓		
English Literature. English Language. Fr. Mathematics. History. Religious Knowledge. Art.	Os ✓			✓
State Registered Nurse. State Certified Midwife. Psychiatric nursing. Health Visitors Certificate	✓			✓ (42, 43)

) Please state why you have chosen to study D303?

Continued interest in psychology.
Possibility of obtaining work in area of clinical psychology when family old enough to leave.

(b) If you have given several reasons above, which is the most important to you?

first

(44, 45)

6. Please ring code 1 for whichever of the following statements is true of you. *Ring more than one code if more than one apply*

I have no idea or little idea of what the D303 Artificial Intelligence project will involve	(1)	(46)
I have no knowledge or little knowledge about Artificial Intelligence	(1)	
I've heard about the Artificial Intelligence project from other students	1	
of my reasons for doing this course is the opportunity to use the computer	1	
I look forward to using the computer	(1)	(50)
I find the prospect of using the computer daunting	(1)	
I don't think the use of computers is relevant to the subject of Psychology	1	
I am looking forward to working on the Artificial Intelligence project	(1)	
ave some understanding of what the Artificial Intelligence project involves	1	
I believe I have moderate or good knowledge about Artificial Intelligence	1	
I don't like the idea of using the computer at the study centre	1	(56)

7. If you were telling someone about the D303 course, someone who knew nothing about computers but a little about Psychology, how would you. . . ?

(a) Briefly describe the Artificial Intelligence project to them.
Please read (b) overleaf before answering (a).

An opportunity to explore for oneself the basics of an interaction with an example of artificial intelligence

Briefly describe Artificial Intelligence and its role in Psychology.

Attempt to follow the cognitive processes of human brain by its simulation and study i the form of computer 'intelligence'. Central role in fields of experimental & developmental psychology

Have you used a computer of any kind? Yes 1 / No (2) → Q11 (57)

IF YOU HAVE USED A COMPUTER (Q9 and 10)

Please state briefly the type(s) of computer(s) you have used and the extent of your computing experience.

Please ring one code for each statement below to indicate how true it is of you.

	Frequently	A few times	Never	
a) I have used a computer interactively (i. e. from a terminal)	1	2	3	(64)
b) I have used the following programming languages:				
BASIC	1	2	3	
FORTRAN	1	2	3	
COBOL	1	2	3	
Any other languages (Please specify)				
_____	1	2	3	
_____	1	2	3	
_____	1	2	3	(70)
c) I have used a calculator	1	2	3	
d) I have used a typewriter	1	2	3	(72)

Please indicate to what extent you agree with the statements below.

"Computers are beneficial to society . . .	Disagree	Mildly disagree	Neutral or Don't know	Mildly agree	Agree	
a) as an efficient labour saving device for solving problems"	1	2	3	4	(5)	(73)
b) for extending human capabilities"	1	2	3	(4)	5	
c) being essential to the functioning of modern society"	1	2	3	(4)	5	
d) as information handling devices"	1	2	3	4	(5)	(76)

Please comment on your answers to Q11:

Pretty uniformed opinions at this stage (!
D303 before actually encountering the computer
terminal.

(77,78) 01

CARD 2 (1-23)

(24,25) 01

2. Please ring a code for any of the following 'benefits' you think you will gain from doing the Artificial Intelligence project:

Ring more than one code if more than one apply

(a) Knowledge of computers	(1)	(26)
(b) Knowledge of programming	(1)	
(c) A better appreciation of Artificial Intelligence work	(1)	
(d) Opportunity to carry out Artificial Intelligence type(s) of work	(1)	
(e) Any other benefits - please list below	(1)	(30)

perhaps it will help me to think in an organised, logical fashion.

3(a) The Artificial Intelligence (SOLO) TMA in D303 is compulsory. If it were not, would you do it? (31)

Yes (1)
No 2
Don't know 3

(b) Please comment on your answer to Q13a.

This is probably true as I like a challenge, and haven't avoided them in the past except - I did not use a computer in D305 because of physical factor preventing the travel involved last year.

THANK YOU FOR YOUR HELP IN COMPLETING THIS FORM. I HOPE TO FOLLOW UP THE RESULTS SO YOU MAY BE ASKED TO HELP AGAIN IN THIS WAY LATER IN THE YEAR.

ANN JONES
STUDENT COMPUTING SERVICE

Appendix 7.2 Final Questionnaire

Microcomputer Questionnaire

1. Below is a list of terms and concepts which you encountered in the microcomputer experiment book. Please explain each of them as you would to someone who knows virtually nothing about computers. Write at least one sentence on each, and describe them in any way you like, for example, by explaining what 'it' does, or how it works (if relevant), or you could explain the terms with reference to others.

Program

The task given the computer to perform & the method used to get the result

RAM

I can not remember but I think it was a type of program.

Accumulator

A part of the memory which holds the information of one address, & to which numbers can be added.

Single-step

A key which adds a single digit to the address & brings the number at that address into action.

Address

A section of the program memory which contains a number.

Program memory

The data which has been fed into the program & is held & is accessible when needed.

Data

information fed into the computer

Register

Similar to address but a separate system.

1. Continued.. . . ..

Data memory

the data fed into the computer which can be stored & recalled later.

Program counter

The JIFO key

One of the keys which alters the program.

ROM

As for RAM.

Subroutines

Routines within the main program.

Address pointer

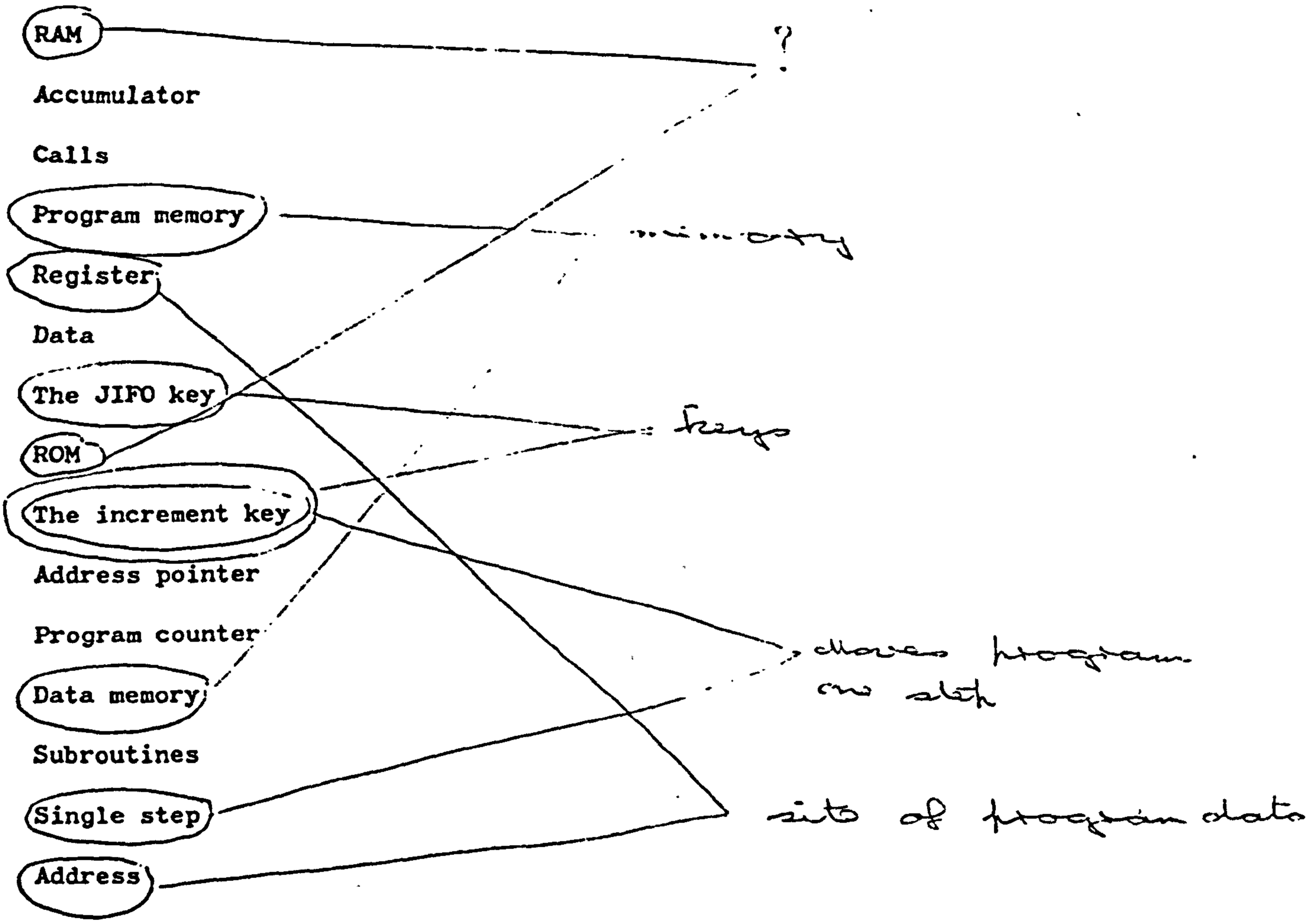
the number which indicates the address

The 'increment' key

moves from one part of the program to the next.

2. Do any of the aboce seem to you to 'fit together' in any way? If so, please ring the appropriate terms and link them with a line and explain how they're connected.

The terms have been repeated below for you to ring them.



3. You worked through the microcomputer book at roughly the same time as you did the SOLO Artificial Intelligence TMA. Please tick whichever of the following is true, and comment on your answer.

✓ 1. I worked through the microcomputer exercises before doing the TMA.

2. I had finished the TMA before I started working with the microcomputer.

3. I was doing both the TMA and working through the microcomputer exercises during the same time period.

4. Other - please specify.

4. Please comment on the following:

1. The things about the microcomputer which have nothing in common with SOLO.

2. **The things about SOLO which have nothing in common to the microcopmuter.**

Solo use words & the microcomputer uses numbers. The program are quite different eg traffic light & room temperature

3 **Anything which was common to both the microcomputer and SOLO, or any similarities you noticed.**

Data base or memory.
If you made a mistake it is sometimes difficult to find it & one has to start the program again.

5. **The microcomputer is very different from SOLO, - did you feel that working through the exercises:**

1. helped to clarify what you'd learnt in SOLO?

✓**2. made no difference?**

3. was confusing?

Please explain your answer.

Because they are different it seemed quite separate, but I think it helped me to understand computers a little better.

6. **If you have any other comments please put them in the space below. Thank you.**

I enjoyed working through the exercises, but they took longer because I had to repeat some before I could go on to the next. There are a lot of the terms which I have forgotten.

Appendix 8

Appendix 8.1: Concepts introduced in the Open Logo tutorial manual

Appendix 8.2: In text questions from the Open Logo tutorial manual

Appendix 8.3: Evaluators' reports

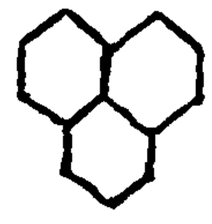
Appendix 8.1: Concepts introduced in the Open Logo tutorial manual

Appendix 8.2: In text questions from the Open Logo tutorial manual

LOGO Questions	Context/Chapter
Q1.1 Can you think of a (p17) more systematic way of drawing a square? Can you also think of a way to use the copy Key to get your instructions to the turtle?	Sect. 1.4 Exploring with the turtle the commands DR FD BD LT +RT A sequence of instructions has been typed in and executed: FD 200 RT 90 BD 200 LT 90 etc...
Q 1.2 How many times does each of the lists need to be repeated? Why? (p21a)	Sect 1.5 The command REPEAT Try drawing triangles and by incorporating the lists of instructions. FD 200 RT 120 or FD 100 LT 144 into a REPEAT instruction.
Q 1.3 Can you work out what these are? Write down the commands that will draw on appropriate circle, and your explanation (p216)	Sect 1.5 It is possible to use REPEAT and FORWARD commands to approximate circular motions.
Q 1.4 Try to predict the effect of the command REPEAT 8 [RT 45 REPEAT 4 [FD 50 LT 90]] (p22).	In fact the REPEAT command is very powerful as it will repeat anything including other REPEAT instructions.
Q 2.1 Can you use ARCR instructions to draw petals, flowers and borders? (p29)	Section 2. Drawing turtle graphics 2.2 Curved lines: the commands ARCL and ARCR
Q 3.1 Construct a procedure to draw a HEXAGON (six sides) hint you only need to use the commands FD RT REPEAT. (p38)	Section 3. First LOGO procedures. Now you can use the To command and editer to construct programs for any of the procedures that you have already met, e.g. To Star PENDOWN REPEAT 5 [FD 5 RT 144]

Q 3.2 Can you modify HEX to do this? (p42)

3.5 Creating longer programes
Let's change HEX into a program that draws three hexagons, like this:



(HEX is defined as REPEAT 6 [FD 50 RT 60]

Section 4 variables

4.1 Creating procedures with variable input.
SQUARE SIDE has been defined:
PENDOWN
REPEAT 4 (FD Side RT 90)

Q4.1 Using the procedure SQUARE can you draw smaller squares with sides of length 50, and 10 and a larger square with side length 200? (p47)

Logo knows how to interpret this as an instruction to use the SQUARE program with the name SIDE replaced by 100. It draws a square with each side 100 turtle units long.

Q 4.2 Can STAR be used to draw any, regular polygon? What values will SID) and ANG need to draw: a 5 sided figure, a 7 sided figure, a 14 sided figure? How would you work out the values? (p49)

Sect 4.2 Procedures that use several variables.

The program
STAR SID ANG
REPEAT 1000 [FD SID RT ANG]
is defined.
Experiment with instructions such as: STAR 40 90
STAR 100 120
STAR 60 144
STAR 200 178

Q.4.3 Can you modify STAR so that it it will now have three variables: NUM (Number of sides) SID (Side length) and ANG (angle).

Perhaps you should also change its name to POLYGON to make it more accurate.

POLYGON NUM SID ANG

Should be able to draw any polygon (with the right values!) (p50)

Q.4.4 Can you further modify POLYGON (or create a new procedure called something else) which has one variable (NUM - number of sides) as its input and will draw a polygon with that number of sides? e.g. POLYGON 4 should produce a square, typing POLYGON 3 should produce a triangle etc. (p50)

Write all your attempts and comments on your comment sheets. (No answers are provided for Qs 4.3 and 4.4)

Question 4.5

Can you now construct a procedure FLOWER which has 6 Petals and uses the PETAL procedure? Record all your attempts (and comments) on your comment sheets.

When you've successfully defined FLOWER, construct a procedure BORDER which gives you a border of flowers. (page 51)

4.3 Modular programming

"When we constructed procedures to draw petals, flowers and borders we quite naturally described each in terms of the previous one. LOGO helps you to incorporate this natural structure into your programs.

Once a procedure has been defined it can be used as a module of a larger program in exactly the same way as if it were a primitive command."

The procedure PETAL is defined:

```
PETAL R

REPEAT 2 (ARCL 100 60 LT 120)
```

Q 5.1 Try it (p58)

Sec. 5.1 Tail recursion

The following program has been constructed:

```
RSTAR SID ANG
FD SID
RT ANG
RSTAR SID ANG
```

... You should be able to turn the simple iterative procedure:

```
BOX
REPEAT 4 [FD 40 RT 90]
```

into a procedure Box that uses tail recivision

Q.5.2 Can you think of a way to modify the BOX procedure that results in it drawing more and more boxes each half the size of the previous one?

(LOGO can use the symbol '/' to mean divided by and so /2 can be used for half) p62

(unnumbered in-text question)

What happens if you type PRINT 'SID? Why? (p66)

Q 5.3 What would happen if you typed in RECUR 200. Why?

Section 5.2
Varying the variables.
The program

```
SPIRAL SID ANG INC
FD SID RT ANG
SPIRAL SID +INC ANG INC
```

has been defined and used.

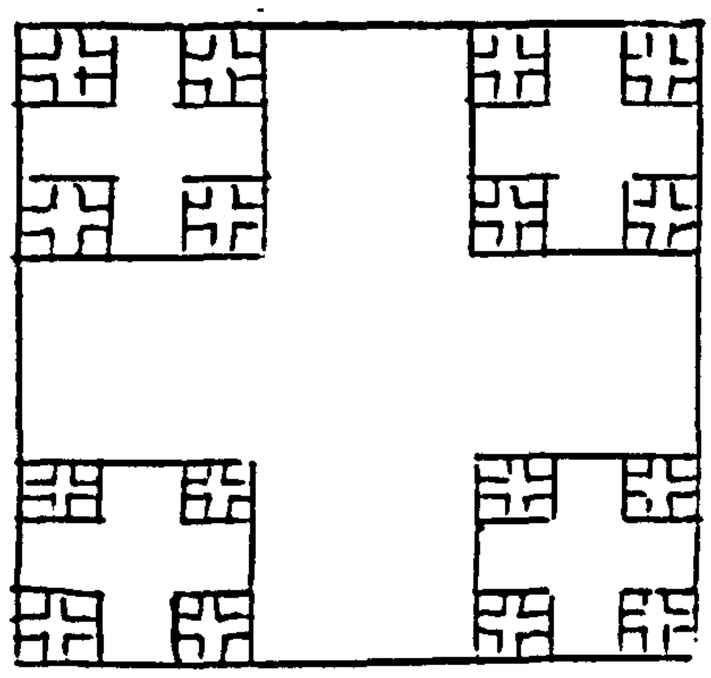
```
R BOX
FD 40 RT 90
R BOX
```

and

```
BOX
REPEAT 4 (FD 40 RT 90)
```

Section 5.4
The command MAKE
In MAKE 'SID 20 R
The quotation mark tells LOGO that it is the thing named SID that is to be given the value 20. If you leave it out LOGO will look for a value for SID and unless that value is a suitable name for a variable, an error will result. You can use the PRINT command to check what value a variable has, Try PRINT SID.

Sect 5.7 Total recursion
"recursion can be used to perform the same (or nearly the same) instructions, over and over again .... this has not revealed the power (of) being able to refer to a procedure itself from within the program actually defining the procedure. This is total recursion



"(It can be described as)...
a pattern within a square and at each corner is a similar pattern ½ the size"

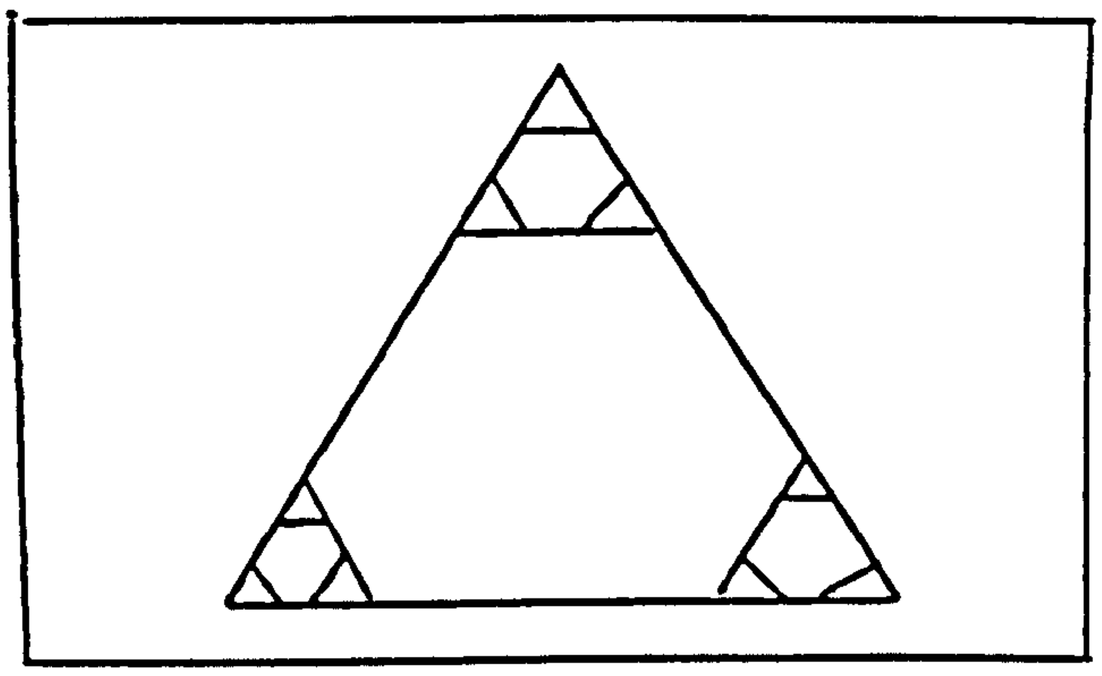
Create

```
RECUR  SID
REPEAT 4
(RECUR SID/3
 FD SID RT 90)
```

Did you predict how the pattern would emerge? If you did, then you are well on your way to understanding recursion.

Question 5.4

Can you adapt this recursive procedure to obtain one that draws this pattern made up of equilateral triangles?



(p75)

Section 5.7
the program RECUR is defined:

```
RECUR SID R
REPEAT 4
(IF SID <20
(FD SID RT 90)
(RECUR SID/3 FD SID RT 90)) R
```

"type in the instruction

```
RECUR 200 R"
```

Appendix 8.3: Evaluator's reports

Evaluator 1

(The main comments were written on the front of the tutorial manual and are reproduced below).

1. Keyboard familiarity - is it presumed?
2. Like the friendly reassuring tone of 'tutor'
3. What about a quickie intro to 'Logo' machine?
4. I missed a global explanation in breif at beginning of some sections (I have marked these with an asterisk)
5. Order sometimes puzzling - intro of lists a bit late? Other things marked inside
6. Recursion is just not easy to explain! I haven't got a constructive contribution to make to that one

Evaluator 2

To: Ann Jones **Date:** 8.12.87

Subject: Comments on OPEN LOGO tutorial manual

(Ann, sorry this is late!)

General points:

(1) You need to explain the differences between commands that appear to be similar (because of their names or what they do) - e.g., ERASE, DELETE, CLEAR, BLANK, etc. The reference manual probably does this, I don't know, but the tutorial made it confusing for me! Some sort of summary, giving a "model" for each command, and grouping them according to functions, would help.

(2) I like the introduction to recursion - wish I'd had it when I was learning Lisp!

(3) If you're assuming that students have a non-mathematical background then you'll have to explain some of the terms in chapter 7 (e.g., some of the trigonometry), even if only as a "refresher".

Detailed comments: (I've also marked these on the relevant pages of the manual.)

(Page 7) The sentence *"The chapters and sections are not intended to indicate individual study sessions; you should work through them at your own pace."* This is confusing... Better if you make it an explicit instruction.

(Page 9) I don't know whether there's a separate tutorial for setting up the computer, but throughout this tutorial you refer to the reference manual as if it's to be used after going through the tutorial. However, the first sentence on this page seems to assume students have already encountered the reference manual.

"...if there are chips with a higher precedence" what does this mean??

If you're referring to how the screen looks, you should show it with a diagram (e.g., *"The right-pointing arrow on the second line of the screen is the Logo....)*

You talk about moving back and forth between Logo and Basic, but I thought you were

assuming that these students had no programming background...!
(Page 10) The diagram of the keyboard is a bit confusing - I would show the whole thing rather than bits of it, so that keys can be clearly identified.

Why do students have to use caps?

(Page 11) Inconsistent use of terms "diagnostic message" and "error message".

What is "teletext mode"; will students understand this? The whole paragraph at the end of the page is rather confusing.

(Page 12) What does COPY do? It doesn't seem intuitive - i.e., how does it "reproduce the remainder of the line". (See my general comments above about explaining how commands work.)

WIPE: what does it mean for everything to "vanish" - is it deleted?

I think the term "mode" should be explained - especially if these students are computer naive.

(Page 20) The discussion of "physical lines" versus "logical lines" is not very clear.

(Page 23) How does RUBBER work? What does WRAP do? (Again, see general comments above.)

(Page 24) How come things are speeded up when you use HIDE?

(Page 26) In explaining what the pens do, you should make it clear at the top of the page that pen 0 is for the background.

If you're going to mention logical versus physical colours they should be explained here, even if only briefly.

(Page 28) How does ESCAPE work? When is it better to use FENCE and when ESCAPE?

(Page 31) It's not clear whether or not you mean the students to draw the picture in the diagram...

(Page 40) The concept of a buffer should be explained (cf. "print buffer).

(Page 41) You should explain what happens in renaming (e.g., in terms of making copies of files, etc.)

(Page 42) Explain why continually entering the editor destroys any pictures you may have created - a bit confusing, that sentence.

You may have to explain what variables, values and parameters are...

(Page 53) How does MAKE work?

(Page 54) Although you point the student to the reference manual for an explanation of data types, I think you should give at least a brief explanation here, if you're going to mention it at all.

(Page 61) What does *"the 'default mode' of your filing system"* mean?

(Page 62) This is a criticism of the software, rather than your manual, but most commands have fairly mnemonic names - CAT definitely isn't one! Perhaps explain? (I just realised, I have assumed this means "conCATenate" as in UNIX, but I guess it might mean "catalogue"?)

(Page 63) How does SCRAP work?

Why does DURRY appear twice in one command line?

What is the "REMOTE facilitity"?

What does COMPACT do?

Appendix 8.3: Evaluator 3

1. Background

I would like to point out that I am writing this as someone who is a regular user of Open Logo. I teach a course on computing to adults with learning disabilities. I think that gives me quite a valuable viewpoint because I have experience of Logo users who do not have any related background, and about whom few assumptions can be made. I feel that I should also point out that, as a practitioner I had *not* used the tutorial book. I think that a fairly cursory inspection and occasional attempts to use it as a reference had suggested to me that it was not very helpful.

2. Overview

It seems to me that any written work involves a number of participants:

the writer,
the reader,
the reader, as perceived by the writer.

Notice that the last two *may* not coincide. The challenge of writing is for the writer to communicate with the reader, based on their perception of the reader. So, there are two potential sources of error for the writer: that they do not communicate well, or that they communicate with the 'wrong' person - or both.

One aspect of the writer's perception of the reader is that the piece of writing is likely to be targeted at a particular audience. That audience will be defined broadly in categories such as: children, experienced programmers, doctors etc. In other words, the writer will make certain assumptions about the background of the reader, about their experience. Problems occur when the writer's assumptions are wrong, either explicitly because a reader comes from a background other than that targeted, or because the writer has made unconscious assumptions. This book suffers from both defects.

Firstly, it is not clear whom the intended audience was; whether it was school children learning Logo, or adults (Open University students, perhaps). In some places I found its tone patronizing, which suggests it was written with children in mind, but even so would be unnacceptable. The author should be told that putting an exclamation mark on the end of sentences does *not* make them chatty and friendly. As far as I am concerned, it merely annoys.

The second error was made also in that the writer has often made assumptions about the meaning of words. In his (and I believe the authors were both male) world some words have quite specific meanings which are well understood, but they do not have the same meaning in common speech. He has not stopped to think about mis-apprehensions he might have caused.

Edsgar Dijkstra said, "The use of anthropomorphic terminology when dealing with computing systems is a symptom of professional immaturity". He was perhaps being a little hard, but the sentiment is a correct one, and I believe anthropomorphization occurs too frequently in this text. To use a word such as *think* implies a great deal, regarding such matters as consciousness and self awareness. Thus, though it may be convenient to write phrases such as *the turtle knows...* (page 28) it is certainly untrue, and may be misleading.

I have observed that experienced computer users have developed a healthy cynicism regarding documentation; they know that it is often plain wrong, but novices tend to expect the same fidelity they would find in the printed word elsewhere. An inaccuracy in a text for novice users can therefore be most damaging. The reader is likely to lack confidence in their own ability and judgement and if anything happens on the computer which does not correspond with that which the text told them to expect, then they will assume it is they who are at fault; they would not believe the text could be wrong. Thus any such inaccuracies are most serious, and I am afraid this tutorial contains a significant number of them.

3. Detailed comments

Chapter 1

I thought this was the worst chapter. This is unfortunate as it is likely to put readers off, who will venture no further, and so never read the better subsequent chapters. I suspect that it must have been written by a different person from the other chapters. I also suspect that it was written before the language implementation was complete, as it contains some errors in respect of the way things work.

Page 9

...if there are chips with a higher precedence...

This phrase, in the second paragraph, is likely to strike fear into the heart of the timid newcomer. Surely it could have been written as something like, "Depending on how the Logo chips have been installed, you might not have got that message..."

*>*LOGO*

Strictly speaking, if you read the instructions, it should be clear that the user does not type the >, but I think it would have been worth stating explicitly.

That's all there is to it.

To what?

So stick with Logo!

An unnecessary exclamation mark, and a silly sentence, anyway.

Page 10

Figure 1

I thought that this was confusing, because it did not make it clear that it was only illustrating a selection of the keys.

PRIMT 'HO

Two things have changed since the previous example. When I read it I saw only the fact that it said *HO* when previously it had said *HI* and assumed that was the deliberate error. The instructions which followed then become confusing.

Page 11

that is a list to be quoted

That would mean nothing to a novice.

These brackets will appear on the screen as ← and→ since we are in teletext mode.

Three things wrong with this sentence. Firstly it is wrong. If the user

follows instructions, then the brackets will appear as brackets. Secondly even if they did appear as arrows, the typeface is so different that they would not look very much like the the arrows in the text. Finally, what does it mean to be in teletext mode? Why confuse the reader with such jargon?

Page 12

If the reader follows the instructions exactly as laid out they get a line reading *>PRINTT*. It is inexcusable to so confuse the reader at this early stage. Also, I feel that the copy facility is introduced too soon, that it will confuse the student unnecessarily. At this stage the amount of effort involved in re-typing their input would be small.

I do not think that the role of lists has been made clear enough for the reader to be told to use them in print statements. It is not clear when to use a list and when to use a quote.

The BBC Microcomputer has eight distinct modes...

Here is an example of unnecessary and confusing use of jargon. I think the word *mode* should have been avoided as far as possible, and at this stage the student could simply be told that *MODE 1* is just something they must type, and they can learn elsewhere what the implications of using it are. Also the word *mode* is used elsewhere in different ways. Already there has been mention of *teletext mode* (see above).

Page 13

...takes you from the Logo text mode... to the Logo drawing mode.

"Oh, these must be two more of the eight modes mentioned on the previous page." thinks the reader. Wrong.

...the screen springs to life

No it does not. It merely goes blank and then displays a frame. Anyway, I do not find such terminology chatty, as I suppose it was intended to be.

where Logo will give its responses to your instructions

I would prefer: *where Logo will give its printed responses to your instructions* since drawing in the drawing window is also a response.

This is the turtle!

Ugh!

Page 14

i.e. turns it to its right

This is most ambiguous. Of course it turns to its right, the instruction mentioned the word *right*. What I assume the author meant was that a person regards anything 90° from the direction they are facing to be on their right side, but I think saying this just confuses the matter.

Page 18

Figure 10

A problem with all the figures in this chapter is that they are distorted. I

assume that this is because they have been taken from screen dumps of low quality displays. Thus, the shape in Figure 10 is clearly *not* a square, but a rectangle, but no explanation is given of this. This will confuse the reader who can clearly see than in their program the command to draw each of the sides was *FD 200*, so that they would expect all the sides to be equally 200 units long. Either the reader should have been told something along the lines of: *You may notice that the shape on your screen and the one in Figure 10 do not appear to be squares, that their width is greater than their height. This is because of the limitations of computer displays*, or the illustrations should have been corrected, and the reader warned that the pictures on their screens might appear distorted.

Page 19

Figure 12

Another inexcusable error. The diagram is wrong. The program as given would print a left-handed star. Any poor, novice student who got something which did not look like Figure 12 would assume the fault was theirs.

Since Logo is a sensible language...

A meaningless and somewhat patronizing phrase; looping is a fundamental feature of all programming languages - 'sensible' or otherwise.

FD 100 LT 144

Any learner looking at examples needs to be able to spot what are the important features of each example. This is aided if each example builds on an earlier one, and if the only difference between successive examples focuses on the point of interest. In this case all previous examples have used the same distance, 200, as the parameter of their *FORWARD* instructions. Suddenly here is one example where 100 is used, and the reader is left to wonder whether some property of pentagrams is not only the angle of 144, but also the length of 100.

Page 20

Figure 13

Again, the reader has been told to draw a circle, but is shown an elipse.

...and don't worry that your typing goes on to a second line on the screen

I think this reassurance comes too late. The reader has already been told to type lists of instructions which would have gone over the line, and should have been warned then.

The turtle draws Figure 14 in a most interesting way.

I don't know what is interesting about it.

Page 22

I think the reader should be warned that they will get an error message when they try to get the turtle to cross the 'fence'.

Chapter 2

As I have suggested, this is not as bad as Chapter 1, so there are fewer comments here.

One of the features of Logo which I like least is the significance of the ends of lines. The authors of this book have tackled this problem by trying to show explicitly where ends of lines occur. However, there can be some confusion in the way they have done this because there may be a suggestion in the reader's mind that where the end-of-line symbol appears they must explicitly type a carriage return, and that is not always true.

Page 24

Figure 21

This diagram is very unclear. It contains two boxes but it is not evident what they represent. Is the outer one the screen and the inner one the drawing window, or is the outer one the drawing window and the inner one a rectangle drawn within it? In either case, the diagram does not represent what the student would be observing on their real screen.

Page 25

BD 50 LT 90 FD 50 RT 90

This is an example of the problem of ends of lines. None is shown on this line, and none is necessary, but the reader may wonder how the next statement got on the next line without a carriage return. In fact, they would have to type a large number of spaces to achieve that. The same is true in the example on the following page, just under Figure 24. In fact, in this case no carriage returns are shown, so strictly speaking nothing would happen.

Page 28

I think there are too many sentences with exclamation marks on this page!

Pages 30-32

The table on page 32 partly repeats the one on page 30, which I think is unnecessary. Also, I think the discussion of colours is confusing. The student is told that there are 16 colours, then that there are only 8, and then is given a table of 16.

Chapter 3

It is mainly this chapter which suggests that the tutorial was written before the system was fully implemented, because the description of the operation of the editor does not coincide with the reality.

Page 33

There is confusion here. The diagram clearly shows the title line of the function and the cursor on the line below - which is indeed how the editor works. However, at the bottom of the page is the instruction: *Press the RETURN key and the cursor moves down to the beginning of the next line where we shall begin typing the instructions for our BOX program.* Wrong.

Page 34

f6 can be used to delete to the right

In fact f6 deletes the character at the cursor position.

Page 36

In all the examples of defining functions in this chapter it ought to be made quite clear what the user must type and what is already entered in the editor for them. Specifically, when the editor is opened to create a new function, the name of that function appears on the top line. It must be there, but the user does not need to type it. However, for example, the definition of STAR seems to imply that the user must type the name STAR. This mis-apprehension is reinforced by the fact that the line containing the word STAR is shown with a carriage return character. Strictly speaking that is correct in that there must be one there, but in fact the editor inserts it automatically and the user does not type it. If, as is likely, the user types everything shown in the example then there will be two lines in their function containing the name STAR and the second one will be treated as a recursive call: infinite recursion. I know of one student who actually did this.

Further down the page the student is told to enter MODE 1, followed by STAR. Up to now the student has been told to type DR (or DRAWING) between these two instructions. In fact it is not necessary so to do, but the student might now be confused by the absence of that instruction.

Page 37

Instructions to EDIT BOX are shown three times. I think the reader might be confused about how many times they must type it.

Logo does not know what to edit and is confused

Anthropomorphization.

Page 38

Use the down-arrow key to move the cursor down so that it is now positioned under the F

As discussed above, this is not necessary, the editor will start up with the cursor already in that position.

Page 40

Press the COPY key.

I would prefer *Press the COPY key* once. The reader might be confused with the use of the COPY key outside the editor, when it must be pressed once for each character to be copied.

Chapter 4

Page 42

Here again we have the problem of distinguishing what the student must type and what is provided by the editor. There is again the danger of an inadvertent recursive call. Things get more confusing when parameters are introduced because now the programmer must move up to the function title line in the editor and and the parameter name.

Page 46

More problems with carriage returns and line ends. I wonder whether this was a bit early to worry the student about laying out their functions, and

the distinction between physical and logical lines. I think mentioning layout is probably not worthwhile since the system contracts multiple spaces anyway. That means that having gone to the trouble of laying out a function, the next time the student edits it they will find all the formatting has disappeared. I really think they should be told so at this point.

Chapter 5

I like the fact that recursion is introduced at an early stage in this tutorial, since it is an important feature of Logo programming. I do *not* like the way that it is introduced by examples involving infinite recursion, which has to be interrupted using the ESCAPE key. This gives the impression of recursion being something out of control.

Page 48

The drawback with iteration is that we need to know beforehand how many times to perform the same repetitive task...

I would prefer *The drawback with* REPEAT *is that we need to know beforehand how many times to perform the same repetitive task.* It is *not* true of iteration in general. For instance a WHILE loop may terminate on a condition which has nothing to do with the number of iterations.

So now they introduce recursion, which does not have this 'drawback', but how is it introduced: as a mechanism whereby the task is not repeated some fixed number of times, but an infinite number of times, until the user hits a 'panic' button.

Page 49

but clearly SID + INC must qualify

"Clearly" to whom?

Page 51

Somehow the use of the division operator, '/', is introduced twice on this page.

IF SID ' 20

Surely some mistake here. *IF SID > 20* perhaps?

else stop

This use of *else* is how programmers talk. Other people would probably say *otherwise* and I think that would be better here.

Page 54

MAKE 'MESSAGE 'HI

I think it is unnecessarily confusing for the first example of assignment to be of a string value. The two quotes in the example are doing different jobs. I'd stick to numerical assignments for now and do strings later.

Page 58

Don't forget to colour your pictures!

Was this tutorial really written for three-year-olds?

Page 60

The turtle is forever thinking...

Oh, no it isn't.

Chapter 6

Page 61

...you should make a manual record of where on the tape this file has been saved.

...using the tape counter.

Page 62

...or you are running short of memory

How would the user know? They might like to be told of the symptoms (i.e. error messages).

Pages 63-64

Instructions are given for loading from tape and some mention is made of using discs, but no details are given. Keep it simple, fair enough, but some indication should be made (for instance a reference to instructions elsewhere) for those who are interested.

including the !BOOT file

The what file? This will mean nothing to the novice.

*type *BASIC*

Why? This is a course on Logo, not Basic. I think the reader ought to at least be told that it's just a feature that it is safer to do a *COMPACT from Basic.

Chapter 7

Page 65

PRINT SUM 1 1

There is a formatting problem. One way or another, I think it should be made as clear as possible that there is a space between the two 1s, otherwise the reader might read it as eleven, which would make no sense.

Now we seem suddenly to be treated as infant school children. We start being lectured about sharing out eggs. I think we can be treated as grown-ups, just use the word REMAINDER straight out.

Page 66

MAKE 'D A + A

Again I think it is important to stress that there is a space between the *D* and the *A*..

Page 67

...or you can challenge a friend...

Is this a tutorial or the Beano? Confusion again about the age of the intended readership.

Page 68

The example of the guessing game leads the reader through the player getting the answer wrong, but then shows them what would have happened if they had got it right before what would actually happen in this case. I think that should be reversed.

Page 71

...given one value, returns another related value...

This is using the word *returns* in the way a programmer understands, but which is meaningless to most other people. The process of returning a value should be explained.

Is the table of functions, including esoteric concepts such as arctangents and natural logarithms really intended for the same audience as were being told about remainders with the help of eggs a few pages earlier?

4. Conclusions

Well, now I think I know why I did not use this text. It appears to me to have been written in a hurry, so that it was not as well thought out as it might have been, and not checked as thoroughly as it ought.

Sorry if the comments got a bit wild near the end. I was writing them at night and nearly got carried away.

Appendices

Appendix 9.1: Examples of the in-text exercises and answers

This appendix contains examples of DESMOND questions and answers drawn from the seven chapters.

Questions

<u>Exercise 1.1</u> (p21) ...move to..address 33 Start to change the contents to 134 by pressing [1], [3] and then [4]. What happens if you now press [->]? What should you press to complete the process of putting 134 into the location with address 33?	<u>Context</u> Address and their contents have been introduced and changing contents in Monitor Mode
<u>Exercise 1.8</u> Change the contents of the memory location from 108 to 107...	<u>Context</u> Altering a program has just been covered
<u>Exercise 1.11</u> Try the temperature and light sensors with the denary display. Alter the contents of address 1 to 107 for the light sensor or 106 for the temperature sensor. What range, in denary, can you obtain from the heat sensor?	
<u>Exercise 2.2</u> a)What is the operation code for 'Jump to the specified memory location?' b) What is the instruction code for 'Jump to address 60'?	
<u>Exercise 2.8</u> Write a program that will take a number stored at the memory location with address 90 and copy it to the lamps. By storing a number at the memory location with address 90 prior to running the program (Using Monitor Mode) you can thus light the lamp in any way you choose. To start with use the number from the last exercise and check that L3 and L4 come on when you run your program. Then experiment with other patterns.	

Exercise 2.16
Write a program that will cause lamps L1 to L4 to come on in turn and then keep repeating. This will make them appear to flash off and on very quickly. Try running the program in Single Step mode so that you can slow down the operation and see the individual lamps working.

Exercise 3.5
Try entering the following numbers into locations 91 to 98. Use the program to discover what the display shows.
56,43,57,32,61,32,49,55

Context
Writing messages on the display

Exercise 4.5
Write a program that will multiply the value stored at address 91 by 16 and store the result at address 92. Do not add the number to itself 16 times, try to think of a neater method.
Try out your program on the following initial values 2,3,4,5,10 and 15 and account for your results.

Exercise 5.8
Write a program so that the Motor acts as a sort of counter. Every time a key is pressed the Motor should move one position

Exercise 6.5
Write a program to count how many times the push switch is pressed and show the count on the display. You will need to keep a copy of the count in some convenient place - say location 90. Do not use JSR 220.

Exercise 7.4
Write a program to demonstrate a digital clock with a speed adjustment. The clock part can be a simple counting program using the denary display. A pause should be used between each increase (or change of the clock)
Answers

Exercise 2.8
Load into the Accumulator a copy of the contents of location 90

:015 090

Store a copy of the Accumulator into the memory location with address 101 :013 101

Exercise 2.16

Here is a simple program to turn on the lamps one after the other

```
LDI 001;        Turn L1 on
STA 101
LDI 001;        Turn L2 on
STA 101
LDI 004;        Turn L3 on
STA 101
LDI 008;        Turn L4 on
STA 101
JMP 000;        Repeat
```

Exercise 3.5

The codes given should result in the following display

8 + 9 = 17

Exercise 4.5

To multiply a number by 16, it is necessary to double it four times. The following program uses location 92 to store intermediate results as well as the final results:

```
00 LDA 091;     Get the number (n)
02 ADD 091;     Double it       (2n)
04 STA 092;     Save it
               ;  2n is now in the Accumulator and at location 92
06 ADD 092;     Double it again (4n)
08 STA 092;     Save it
               ;  4n is now in the Accumulator and at location 92
10 ADD 092;     Double it again (8n)
12 STA 092;     Save it
               ;  8n is now in the Accumulator and at location 92
14 ADD 092;     Double it again (16n)
16 STA 092;     Save final answer .....
```

Problem 5.8

```
00 LD1 255;     Put a test value in keyboard location
02 STA 105
04 LDA 105;     Look at keyboard
06 ADI 001
08 JZ  004;     No key is pressed
               ;Key is pressed
10 LDI 000;     Move the motor
12 STA 103
14 LDI 001
16 STA 103
18 JSR 215;     Pause
20 JMP 000;     Repeat
```

Exercise 6.5

```
00 LDI 000;     Clear count location
02 STA 090
10 LDI 032;     Mask for S16
12 AND 104;     Is it on or off?
14 JZ  010;     Not on, so keep looking
20 LDA 090;     Increase count
22 ADI 001
24 STA 090
26 JSR 200;     and display
30 LDI 032;     Wait for key to be displayed
32 AND 104
34 JZ 010
36 JMP 030
```

Appendix 9.2: An example comment sheet

ection	Critical comments	Time take
ages 21-25 ontinued		
	Answers/comments to exercise 1.4, 1.5 and 1.6	
1.4	Hexadecimal equivalent of 25 is 19 Binary " " " is 11001	½ hr.
1.5	16 = 10 = 10000 24 = 18 = 11000 99 = 63 = 1100011 255 = FF = 11111111 128 = 80 = 10000000 0 = 0 = 00000000	
1.6	255 is the largest number because its binary equivalent is 11111111 and as only zero's and 1's are used then the 8 positions are filled with 1's, if any were filled with zero's the number would be less than 255. If you try to enter 256 Desmond buzzes when the 6 is pressed.	½ h

Appendix 9.3: DESMOND plans

Below is an index of the 21 plans in DESMOND, which are then described and discussed, along with the programs in which they are used.

Index

Plan	Function
1	Display the state of a device
2	Using a device to control another
3	Operating a device
4	Using the ASCII routine
5	Stopping
6	Adding two numbers
7	Count
8	Adding continuously
9	Multiply plan
10	Delay loop
11	Test for numbers other than zero
12	Reading the keyboard
13	Subroutine
14	Clocked dealy routine
15	Moving the motor
16	Masking
17	Shift left
19	Comparing sensitivity
20	Inverting the input
21	Comparing values

Plan	Function	Outline
1	**Display the state of a device**	**1 LDA[Address of device]**
(Sub plan)1b	**Display routine**	**2 JSR 200 or 205**
(Sub plan)1c	**Loop**	**3 JMP 000 [Repeat to update display]**

Note that there are two sub-plans - display routine, (1b) and loop (1c). The address in (1) could be 104,106,107 or 108, and either display routine could be used.

This plan is used in the very first program given in activity 1D on page 25 which is:

```
00 LDA 104
02 JSR 205
04 JMP 000
```

This program uses the state of switches routine at location 104 to inspect the 8 switches. In further activities different devices are displayed by changing the contents of address

01. This plan is therefore given and explored as part of the programs which display the state of the devices (as above) and is analysed in chapter 2 where the rationale for continually repeating the program to update the display is also explained. The notion of abstracting a plan for displaying the state of a device from the programs for displaying particular devices is not, however, made explicit. This is true for all the plans.

2	**Using a device (e.g switches) to control another (e.g. lamps)**	**1**	**LDI 104 (switches)**
		2	**STA 101 (lamps)**
		3	**JMP 000 (Repeat**

The first example given is a combination of the first plan and plan 3 (below) where the state of a device such as the switches is loaded into the accumulator and sent to operate the lamps:

Load into the Acc a copy of the contents at location 104 (switches)	015	104 (LDA 104)
Store copy of Acc into the memory location with address 101 (lamps)	013	101 (STA 101)
Jump to address 00 for next instruction	010	000 (JMP 000)

This plan is a little more complex than plan 3 below, which may be the reason for some of the problems encountered with it. Also plan 2 can be more logically viewed as a combination of plan 1 and 3 : i.e. there is an argument that plans 2 and 3 should have been introduced the other way round. The reason that they have not been, presumably, is that the instruction LDI has not been introduced at this point.

3	**Operate a device**	**1**	**LDI [Some number]**
		2	**STA [Some address]**

The address to which the number is sent (instruction 2) determines the device (e.g 101 for lamps) and the number loaded into the accumulator in 1.will affect which particular device is affected, e.g which lamp is lit. The LDI instruction is introduced in chapter 2 on p94. An example of this plan is given on page 95:

```
LDI 006: 014 006; Put the number 6 into the accumulator
STA 101: 013 101; Copy the accumulator to the lamps
```

4	**Using the ASCII routine**	**1**	**LDI [Address of first bit of data]**
		2	**JSR 210 [ASCI routine]**

The address in 1. is where the first bit of data is stored.
This is introduced in this form in chapter 3, p138: LDI 91

JSR 210

Later an example is given of this plan combined with plan 4. This is used in exercises 3.6 to 3.8.

Exercise 3.8

Plan 1	LDA [ ]	LDI 91
	JSR [ ]	JSR 210
	JMP [ ]	LDA 108
		JSR 200
Plan 4	LDI [ ]	JMP 000
	JSR 210	

5	**Stopping**	**N**	**JMP N**

This is not given as a stopping plan but 88 JMP 088 is explained as follows (p138): "This last line 88 JMP 88 will cause DESMOND to keep repeating the same instruction at address 88. In this way the data to be stored at addresses 91 to 98 will never be mistaken for program instructions."

6	**Adding 2 numbers**	**1**	**LDA N1**
		2	**ADD N2**
		3	**STA LOC1(Version A)**

This plan is given (p154) as part of the introduction of addition on DESMOND:-

00 LDA 91
02 ADD 92
04 STA 93

This program adds two numbers at locations 92 and 93 and puts the answer in location 93.

7	**Count**	**1**	**LDA N1**
		2	**AD1 N2**
		3	**STA LOC1**

This plan is not given in this form. The ADI instruction is introduced at the same time as ADD, and exercise 4.2 involves typing in a program from chapter 1 (in numerical code) and viewing it in assembly mode. The program is:

```
AD1  001
STA  101
```

The count plan (7 above) needs to be worked out for exercise 4.3:

"Write a program that will take the value stored at memory location with address 91, increase it by 1 and store the answer back at the same location."

The answer: LDA 91

```
AD1 01
STA 91
```

is plan 7 above, and a little further on its function is mentioned:

"In longer programs it can be necessary to keep count of how many times a particular thing has happened. The program above indicates how this can be done"

Given the problems that were encountered with the count plan, introducing it explicitly and giving further examples here would have been helpful.

8	**Adding continuously**	**00 LDI 01** **02 AD1 01** **04 STA 101** **06 JMP 000**

This is a particular form of the counting plan as used in exercise 4.2., with a repeat loop at the end. Again it is not given, but forms the answer to exercise 4.3.

9	**Multiply plan** (Exercise 4.4)	**LDA [ ]** **ADD [ ]** **STA [ ]**

This is the same as 6 but a specialised version, in that the number involved is itself, i.e. the same address. It is not given but forms the answer to exercise 4.4:

"Write a program that will take the number stored at memory location with address 91, multiply it by 2 and store the answer at address 92, and show it on the display. (Hint: Multiplying a number by 2 is the same as adding it to itself)"

The answer is:

```
00    LDA  91
02    ADD  91
04    STA  92
06    ADD  92
08    STA  92
10    ADD .....)
```

10	**Delay loop**	**1 LDI [ ]** **2 DEC** **3 JZ 4**

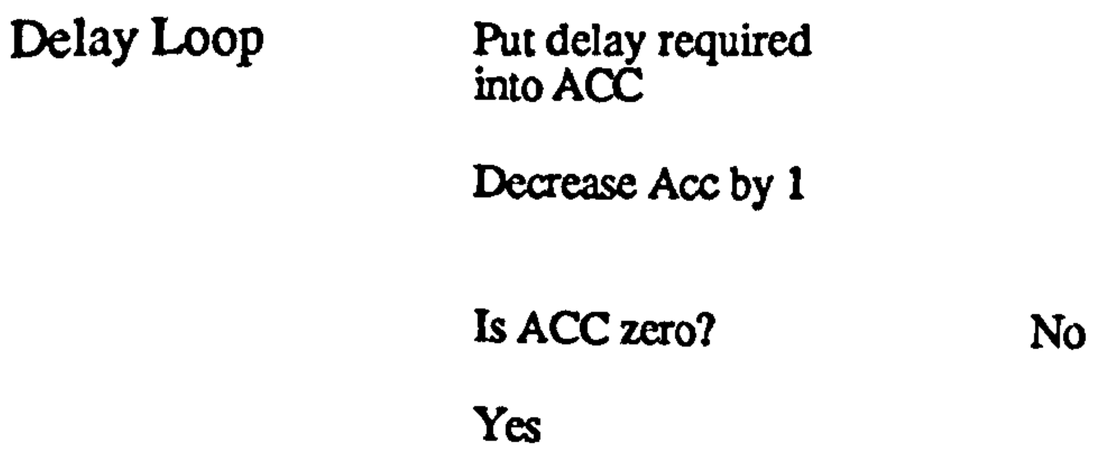
4 JMP 2

In chapter 4, JZ and DEC are introduced, and simple demonstration programs given. It is explained that these programs can be combined to form a delay type loop, and a "typical" example is given on page 165:

```
10 LDI 60;    Load accumulator with length of the delay (60 here)
12 DEC
14 JZ  020
16 JMP 12

20 Rest of the program.
```

The process is explained, and a flow chart is given:



11	**Test for nos other than zero**	**1**	**DEC**
		2	**JZ [ ]**

This plan is not given, but was left for people to work out in exercise 4.7: which most people failed to do. Yet this is perhaps one of the most crucial plans, and was not given. Exercise 4.7 is:

"Write a program that will find the value stored at address 93 and if it is one, light the yellow lamp. This program will use both the DEC and JZ instructions. Use Monitor Mode to store an initial value at address 93 before running the program. Try different values, including zero, to make sure that the program works as you expect it to."

The crucial part of the program is the following instructions which is plan 11 above:

```
LDA 93
DEC
JZ  [  ] (If number now zero, then must have been 1 so branch).
```

12	**Reading keyboard**	**1**	**LDI 255**
		2	**STA 105**

```
3     LDA 105
4     AD1 001
5     JZ  3
```

The first plan given for reading the keyboard is a simpler version of this . It is:

```
LDA 105
JSR 200
JMP 000
```

After this has been tried out, however, a problem is raised:

"How do you know WHEN a key has been pressed? For instance when you run the program above, if the first key you press is the [0], how can you detect this in the program?"

The solution suggested is to put a number in to location 105: whilst this number remains, we know that no key has been pressed, but as soon as a key is pressed the number in location 105 will become one of the numbers 0 to 9 according to which key is pressed. Of course this number must be higher than 9. it is suggested that 255 is used since when 1 is added, this gives zero. This simple program is given:

```
LDI 255;     Put 255 in the Accumulator
AD1 001;     Add on 1
JSR 200;     Display the answer
```

A flow chart is then given of the steps required for a program which sounds the buzzer when a key is pressed. The program itself is worked out as the answer to exercise 4.9. The keyboard plan (12 above) is therefore not given in its entirety. Although the flow chart was given for exercise 4.9, it was found to be difficult (see later discussion). Below is the answer to 4.9 along with the plans needed:

Plan 12	00 LDI 255
	02 STA 105
	04 LDA 105
	06 ADI 001
	08 JZ 004
Plan 3	10 LDI 001
	12 STA 102
Plan 10 (but in different sequence!)	14 LDI 010
	16 JZ 022
	18 DEC
	20 JMP 16
Plan 3	22 LDI 000
	24 STA 102
Plan 2	26 JMP 00

Exercise 4.11, the next programming exercise, uses the following plans:

Plan	Comment
7	Set 'count' store to zero

10	Pause
2	Turn on yellow lamp
12	Read keyboard
11	Test for 1
7	Increase count by 1 (see exercise 4.3)
1B	Display count
2B	Turn lamp off

The 'plans' and their functions are never made explicit in this way and the problems involves modifying and carving up the plans, and were found to be very difficult. And the keyboard routine given in the answer differs significantly from the plan in that a rogue value is not used. This inconsistency is not explained.

13	**Subroutine**	**00 JSR10 (Go to subroutine)**
		02 JMP 2 (Stop)
		10 .....
		12 .....
		16 RET

Subroutines are introduced in chapter 5 and explained in detail. Several examples are given and the importance of the return instruction is stressed. A simple program is given for stepping through:

Main program: -	00 NOP
	02 NOP
	04 JSR 040
	06 NOP
	08 NOP
	10 JSR 040
	12 NOP
	14 NOP
	16 JMP 16
Subroutine:-	40 LDI 015
	41 STA 101
	44 LDI OOO
	46 STA 101
	RET

and further on a more complex version is illustrated in outline:

Main program	00 ----
	...
	06 ----
	08 JSR 040
	10 ----
	...

		16 ----
		18 JSR 040
		20 ----etc
Subroutine		40 ----
		42 ----
		.....
		54 ----
		56 RET (page 192).

14	**Clocked delay routine**	**LDI 000**
		.....
		.....
		JSR 215
		LDI 001
		JSR 215

The clocked delay routine differs from other routines in that other instructions are carried out meanwhile. The plan is given in the context of a simple program to send out a binary sequence to the lamps, making the change on the rising edge of the clock signal:

00 LDI 00;	puts zero into the Accumulator, the initial value
02 STA 101;	sends the Accumulator value to the lamps
04 JSR 215;	wait for the rising edge of the clock
10 AD1 001;	add 1 to the Accumulator
12 JMP 002;	repeat the process

15	**Moving the motor**	**1**	**LDI 000**
		2	**STA 103**
		3	**LDI 001**
		4	**STA 103**
		5	**JSR 215 (pause)**
		6	**JMP 000 (repeat)**

Moving the motor requires the sequence 0,1,0,1,0,1 to be sent to address 103. However, so that the motor has time to respond to one rising edge before receiving the next, a pause is needed between each sequence of 0 and 1. This gives the sequence in plan 15 above, which is given in the text.

16	**Masking**	**LDI [ ]**
		AND [ ]

JZ

This is a process of isolating a single bit form 8 bits. It uses the logical AND instruction. The plan is introduced and given in the context of using a mask to isolate the bit corresponding to switch 16:

```
00  LDI 032;        Load the mask into the Accumulator
02  AND 104;        AND with switch location to isolate S12
04  JZ  010;        zero if S12 is off, come here if S12 is on
06  LDI 001;        put 1 into the Accumulator ready to light the red lamp
10  STA 101;        update the lamps
12  JMP 00;         repeat the whole process
```

17	**Shift left**	**LDA 90** **ADD 90** **STA 91**

Unlike shift right, there is no shift left instruction. To shift a number left by 2 places the number must be doubled. This plan is given (p221):

"If a number we require to shift left is in the Accumulator, the following instructions would perform the task:-

```
STA 099;        Store the number in a temporary location
ADD 099;        Add it to itself."
```

20	**Changing sensitivity**	**LDA [ ]; Load value into Acc** **SHR** **SHR** **SHR** **SHR**

This plan is introduced in the context of a program which reduces the angle sensor's 8-bit range to a 3 bit range by shifting the 3 most significant digits across to the right hand end.

20	**Inverting the input**	**00 LDA [ ]** **02 NOT** **04 .....** **06 ..... **

This involves using the NOT instruction. This, and the above plan are introduced in chapter 7. The program below is given which inverts the value given by the light sensor and so turns the digital light sensor into a digital darkness sensor:

```
00 LDA 107;     Load up the value of the light level
02 NOT;         Invert it
```

```
04 JSR 200;        Display the inverted value
06 JMP 00;         Repeat.
```

21 **Comparing values**

```
LDI  213
STA  90
LSA  108
CMP 90
JLO  50
JZ   60
```

This involves the introduction of the compare instruction (CMP) and another new instruction JLO (Jump if lower flag is set), which instructs the program to branch if the value of the accumulator is lower than the contents of the location with which it is being compared. To explain how both instructions can be used together, which can be quite complex, a program is explained which addresses the question:

"Is the angle sensor value less than, equal to, or greater than 213?"

The outline of the program and branching is as follows:

```
00 LDI 213;        put required test value into location 90
02 STA 90;
04 LDA 108;        Load accumulator with angles sensor value
10 CMP 90;         Compare angle sensor value now in accumulator with
                   test value at address 90
12 JLO 050;        If angle sensor is less than 213, go to 50
14 JZ  060;        If angle sensor = 213, go to 60
        ;          Come here if angle sensor value greater than 213
50.....;           Come here if angle sensor value less than 213

60.....;           Come here if angle sensor value = 213
```

Appendix 9.4: Statistical analysis and quantitative summary

Appendix 9.4 contains a summary of the statistical analysis carried out to test for differences between the scores of group 1 and group 2 on the DESMOND exercises. The tables below give each subject's score for the exercises in that chapter, (where the score is simply the number of exercises marked as correct), and the rank given to each score, and total ranks. The differences between the groups has been tested using a Mann-Whitney test, and the values of U and U′ are also given.

Chapter 1

	Group 1 (N = 10)	*Rank*		*Group 2 (N=9)*	*Rank*
S1	13	13.5	S11	11	6
S2	13	13.5	S12	12	9
S3	12	9	S13	12	9
S4	13	13.5	S14	14	17.5
S5	12	9	S15	10	5
S6	5	1	S16	13	13.5
S7	14	17.5	S17	14	17.5
S8	9	3.5	S18	9	3.5
S9	14	17.5	S19	12	9
S10	6	2			
	T1 =100				T2 = 90

U = 55, U′= 90 (which is non-significant at 0.05%)

Table 9A.1: Differences between groups 1 and 2 in chapter 1

Chapter 2

	Group 1 (N = 6)	*Rank*		*Group 2 (N=7)*	*Rank*
S1	18	7	S7	13	3
S2	17	5	S8	22	13
S3	00	1	S9	21	11.5
S4	12	2	S10	19	9
S5	15	4	S11	18	7
S6	20	10	S12	21	11.5
			S13	18	7
	T1=29				T2=62

U=34, U′=13, which is non-significant

Table 9A.2: Differences between groups 1 and 2 in chapter 2

Chapter 3

	Group 1 (N = 5)	*Rank*		*Group 2 (N=5)*	*Rank*
S1	6	8	S6	5	4
S2	6	8	S7	2	1
S3	7	10	S8	5	4
S4	5	4	S9	6	8
S5	5	4	S10	5	4
		T1=34			T2=21

U=6, U´=19, which is non-significant

Table 9A.3: Differences between groups 1 and 2 in chapter 3

Chapter 4

	Group 1 (N = 6)	*Rank*		*Group 2 (N=7)*	*Rank*
S1	8	11	S6	5	5.5
S2	5	5.5	S7	7	10
S3	5	5.5	S8	9	12
S4	2	1	S9	5	5.5
S5	6	8.5	S10	6	8.5
			S11	4	2.5
			S12	4	2.5
		T1=31.5			T2=46.5

U=8.5, U´= 26.5, which is non-significant

Table 9A.4: Differences between groups 1 and 2 in chapter 4

Chapter 5

	Group 1 (N = 4)	*Rank*		*Group 2 (N=7)*	*Rank*
S1	5	5.5	S5	4	2.5
S2	5	5.5	S6	9	9
S3	1	1	S7	5	5.5
S4	5	5.5	S8	8	8
		89		4	2.5
		T1=17.5			T2=27.5

U=12.5, U′=7.5, which is non-significant

Table 9A.5: Differences between groups 1 and 2 in chapter 5

Chapter 6

	Group 1 (N = 3)	*Rank*		*Group 2 (N=3)*	*Rank*
S1	6	2	S4	8	4.5
S2	6	2	S5	9	6
S3	6	2	S6	8	4.5
		T1=6			T2=15

U= 6.5, U′=1.5, which is significant at 0.05% level

Table 9A.6: Differences between groups 1 and 2 in chapter 6

Chapter 7

	Group 1 (N = 2)	*Rank*		*Group 2 (N= 3)*	*Rank*
S1	2	2	S3	7	3.5
S2	1	1	S4	10	5
			S5	7	3.5
		T=3			T=12

U= 5, U′=5, which is non-significant at 0.05%

Table 9A.7: Differences between groups 1 and 2 in chapter 7

The next table gives the number of errors per chapter for each group and shows which category the errors belong to.

Group 1

	Categories								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Tot. Cats.**
Chapters									
2	1	1			1		5		8
4	4	2				1	1		8
5	2		1	1					4
6	4	1	2		1	1	1	3	13
7	1	2	1						4
Total	12	6	4	1	2	2	7	3	37
%	13.8	6.9	4.6	1.1	2.3	2.3	8	3.5	

Table 9A.8: Number of errors per chapter, and categorisation of errors for group 1

Group 2

	Categories								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**T. Cats.**
Chapters									
2	2	1		1	1		2		7
4	9	4		1		1			15
5	11	2	3	4	2		1		23
6	2	0	3	0					5
7	1		1	1					3
Total	25	7	6	7	3	1	3		52
%	27.6	8	6.9	6.9	3.5	1.1	3.5	3.5	58.4

Table 9A.9: Number of errors per chapter, and categorisation of errors for group 2

Total	37	13	10	8	5	3	10	3	89
%	41.6	14.9	11.5	8.9	5.7	3.5	11.5	3.5	100

Table 9A.10: Categorised errors across both groups

Chapter	No. prog. tasks analysed	No. errors Gp 1	No. errors Gp 2	Total
2	14	8	7	15
4	21	8	15	23
5	11	4	23	27
7	6	4	3	7
Total	61	37	52	89

Table 9A.11 Errors for both groups across all categories

Gp 1	R1	Gp 2	R2
8	6.5	7	5
8	6.5	15	9
4	2.5	23	10
13	8	5	4
4	2.5	3	1
	T^1 26		T^2 29

Table 9A.12 Errors for both groups across all categories, - with ranks and total ranks

	Chapters 1	2	3	4	5	6	7
Gp1	11.1	13.7	5.8	5.2	4	6	1.5
Gp2	10.7	18.9	4.6	5.7	6	8.3	8

Table 9A.13: Mean scores for exercises in each chapter for groups 1 and 2

These differences in table 9A.13 above are not very big. They are not significant, and when the fact that one of group 1 scored 0 in chapter 2 (because she didn't attempt any exercises) is taken into account, the difference almost disappears. In chapters 6 and 7, group 1 subjects attempted few of the exercises.